(4)

Technical Report 1052

Generalizing on Multiple Grounds: Performance Learning in Model-Based Troubleshooting

Paul Resnick

MIT Artificial Intelligence Laboratory

ADA207960

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AI-TR 1052	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Generalizing on Multiple Grounds: Performance Learning in Model-Based Troubleshooting		5. TYPE OF REPORT & PERIOD COVERED technical report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Paul Resnick		8. CONTRACT OR GRANT NUMBER(s) N00014-85-K-0124
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, Massachusetts 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd Arlington, Virginia 22209		12. REPORT DATE February 1989
		13. NUMBER OF PAGES 96
14. MONITORING AGENCY NAME & ADDRESS(If different from Controlling Office) Office of Naval Research Information Systems Arlington, Virginia 22217		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Distribution of this document is unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) none		
18. SUPPLEMENTARY NOTES None		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) learning explanation-based generlization explanation-based learning model-based troubleshooting		

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Model-based reasoning about physical systems has several well-known advantages over heuristic expert systems. These include correctness of conclusions, explanations of conclusions, ease of modifiability and ease of transfer of expertise to new physical systems. On the other hand, reasoning from a model can be slow. This thesis explores ways to augment a model-based diagnostic program with a learning component, so that it speeds up as it solves problems.

Block 20 cont.

Several learning components are proposed, each exploiting a different kind of similarity between diagnostic examples. Through analysis and experiments, we explore the effect each learning component has on the performance of a model-based diagnostic program. We also analyze more abstractly the performance effects of Explanation-Based Generalization, a technology that is used in several of the proposed learning components.

Generalizing on Multiple Grounds: Performance Learning in Model Based Diagnosis

by

Paul Resnick

Revised version of a thesis submitted to
the Department of Electrical Engineering and Computer Science
in May 1988 in partial fulfillment of the requirements for
the degree of Master of Science in Computer Science

Abstract

Model-based reasoning about physical systems has several well-known advantages over heuristic expert systems. These include correctness of conclusions, explanations of conclusions, ease of modifiability and ease of transfer of expertise to new physical systems. On the other hand, reasoning from a model can be slow. This thesis explores ways to augment a model-based diagnostic program with a learning component, so that it speeds up as it solves problems.

Several learning components are proposed, each exploiting a different kind of similarity between diagnostic examples. Through analysis and experiments, we explore the effect each learning component has on the performance of a model-based diagnostic program. We also analyze more abstractly the performance effects of Explanation-Based Generalization, a technology that is used in several of the proposed learning components. [illegible] Explanation based [illegible]
[illegible]
Troubleshooting

Acknowledgments

I would like to thank Randall Davis, my thesis advisor, for his contributions both to this research and to my development as a researcher. His careful scrutiny of all the claims made in this document and his constant admonishments to "drop the shoe" were invaluable.

Numerous discussions with Walter Hamscher helped to clarify my understanding of model-based diagnosis, truth-maintenance systems, and pattern-directed rule invocation. In addition, using some of Walter's code accelerated the completion of this thesis and allowed me to concentrate on the interesting research issues.

Thanks to Janet Kolodner for reviewing the entire thesis and for keeping me honest about the claims that the experimental data support.

Brian Williams has also played an important role in my education, and provided encouraging words when I most needed them.

Guy Blelloch and Eric Aboaf have been willing to discuss all of my whacky ideas about research, politics, and how to structure an organization.

I would also like to thank Phil Agre, Jeff Van Baalen, Meyer Billmers, David Chapman, Choon Goh, Tomas Lozano-Perez, Phyllis Koton, Steven Minton, Paul Rosenbloom, Mark Shirley, Reid Simmons, Prasad Tadepalli, Milind Tambe, Raul Valdes-Perez, Dick Waters, Dan Weld, Patrick Winston, and Peng Wu.

Thanks to my parents for instilling in me the drive to finish this and giving me encouragement along the way.

Denise Sergent helped in many ways. Most importantly, the sight of her face made me smile each day when I went home. She also contributed her editing skills to making the thesis more readable.

The programs described in this thesis used Symbolics' JOSHUA system.

This thesis describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124. Support for the author's artificial intelligence research is provided by a National Science Foundation Graduate Fellowship, and by the Wang Corporation.

Contents

1 Introduction **6**

1.1 Introduction 6

1.2 Summary of Contributions 7

1.3 Multiple Definitions of Similarity 8

1.4 Selecting Useful Definitions of Similarity 9

1.4.1 Utility of an Individual Generalized Rule 9

1.4.2 Aggregate Utility 10

1.5 Performance Effects of EBG 10

1.5.1 Generalizing Successful Patterns of Inferences 11

1.5.2 Generalizing Proofs of Inconsistency 11

1.6 Map of the Thesis 12

2 Similar = Same Contradiction Derivations **13**

2.1 Candidate Generation 14

2.2 The Learning Component 17

2.2.1 Explanation-Based Generalization 17

2.2.2 EBG on Conflict Set Derivations 18

2.2.3 Example: Generalizing the Derivation of Conflict Set (M1 M2 A1) 18

2.3 The Augmented Diagnostic Algorithm 20

2.4 The System in Action: A Carry-lookahead Adder 23

2.4.1 The First Case: 6 + 6 = 10 23

2.4.2 The Second Case: 7 + 14 = 17 25

2.4.3 Summary 25

3 Performance Analysis **27**

3.1 Experiment Description 27

3.2 Performance Results 28

3.2.1 Polybox Experiments 28

3.2.2 Adder Experiments 29

3.3 Case Selection for Experiments 30

3.4 Cost Breakdown 30
3.4.1 Without Generalized Rules 30
3.4.2 With Generalized Rules 34
3.4.3 Cost Differential 35
3.5 Breakdown of Results 35
3.6 Utility of Individual Generalized Rules 36
3.7 Aggregate Analysis: Device Characteristics 37
3.8 Aggregate Analysis: Alternative Diagnostic Engines 38
3.8.1 Pattern-Directed Rule Invocation 39
3.8.2 Using an ATMS 39
3.8.3 Keeping Dependencies Only 40
3.8.4 Reducing Costs of Checking Generalized Rules 40
3.9 Trading Off Precision for Speed 41
3.10 Conclusion 41

4 The Sources of Power in EBG 43
4.1 Generalizing Successful Search Paths 44
4.1.1 Biasing Search Toward Previously Successful Paths 44
4.1.2 Encapsulating Patterns of Operator Applications 46
4.1.3 Finding All Solution States 48
4.1.4 Summary of Problem Characteristics 48
4.2 Generalizing Explanations of Failures 50
4.2.1 Finding the Failure Faster 50
4.2.2 Reducing Search 50
4.2.3 Summary of Problem Characteristics 54
4.3 A Note on Parallelism 54
4.4 Related Work 55
4.4.1 The Causes of Expensive Generalized Rules 55
4.4.2 Operationality Criterion 55
4.4.3 Forgetting Rules With Low Utility 56
4.5 Conclusion 57

5 Similar = Same Fault Hypothesis 58
5.1 Fault Hypotheses 58
5.2 Generalizing Fault Envisionments 59
5.2.1 Utility of EBG on Envisionments 61
5.3 Lifting Fault Hypotheses 62
5.3.1 Utility of Lifting 64
5.3.2 Summary 66
5.4 Exploitability 67
5.5 Related Work 68

5.6 Conclusion 69

6 Extensions: Relaxing Similarity Definitions 70

6.1 Similarities Between Patterns of Inferences 70

6.1.1 Role Equivalent Conflict Sets 72

6.1.2 Same Conclusion; Different Reasoning 73

6.2 Similarities Between Fault Hypotheses 75

6.2.1 Same Misbehavior; Role Equivalent Component 76

6.2.2 Similar Misbehavior; Same Component 76

6.3 Summary 76

7 Conclusion 77

7.1 Draining As Much As Possible From One Example 77

7.2 Multiple Knowledge-based Notions of Similarity 78

7.3 Finding Useful Definitions of Similarity 80

7.4 The Sources of Power in EBG 80

7.4.1 Using EBG to Encapsulate Patterns of Inferences Leading to a Goal State 80

7.4.2 Using EBG to Identify Inconsistent Search Nodes 81

7.5 Conclusion 82

A A Circuit With an Exponential Number of Conflict Sets 83

B More Experimental Results 86

Chapter 1

Introduction

1.1 Introduction

Consider a model-based diagnostic engine. Given a structural and behavioral description of a device, and a set of observed measurements at certain locations in the device, the diagnostic engine identifies components whose misbehavior can explain the misbehavior of the device [HD87, Dav84, Gen84]. Unfortunately, a model-based diagnostic engine does not learn from experience. Given the identical device and observations a second time, it repeats the elaborate causal reasoning necessary to produce the same diagnosis. Yet, if "similar" problems occur frequently, it is of considerable utility for the system to recognize new problems as "the same as" ones encountered before, and then jump to the "same" conclusions, skipping the details of the reasoning process.

Consider, for example, the circuit of Figure 1.1(a) with the observations shown. A model-based diagnostic engine propagates values through the circuit to detect contradictions. In solving the first example, it multiplies 3 and 2 to predict 6 at X; it multiplies 3 and 2 to predict 6 at Y; then it adds 6 and 6 to predict 12 at F. That prediction of 12 contradicts the observation, so the program concludes that M2, M1 or A1 must be broken. It then does some additional propagation of values to conclude that M2 is not the broken component, and finally outputs M1 and A1 as the single-fault candidates. That is, either multiplier M1 or adder A1 alone could, by some misbehavior, account for all of the observed misbehavior of the circuit.

Now suppose the program is given the circuit again, this time with the observations in Figure 1.1(b). The two cases look quite different on the surface, but the answer is the same — M1 and A1 are the only consistent candidates — and there is another interesting similarity: the diagnostic engine propagates values through the same components, in the same order, and detects contradictions in exactly the same places. In other words, the diagnostic program performs the same pattern of inferences in diagnosing the two cases.

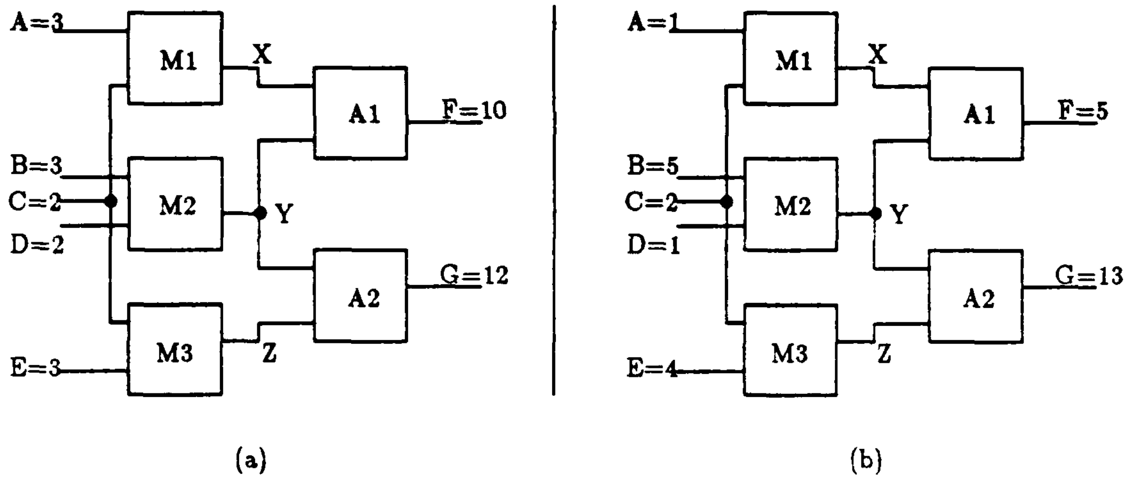


Figure 1.1: The Polybox Circuit, designed to calculate AC + BD and CE + BD. M1, M2, and M3 are multipliers. A1 and A2 are adders. With either of the sets of observations shown, M1 and A1 are the only single-fault candidates.

The augmented diagnostic program described in Chapter 2 can recognize the applicability of a pattern of inferences from a previous problem. It does this by remembering general preconditions for the patterns of inferences it used in diagnosing each case. In diagnosing a new case, it checks each of the remembered general preconditions against the observations. Given the two cases described in Figure 1.1, the program diagnoses the first, generalizes the patterns of inferences that were useful, and, in diagnosing the second case, restricts the candidate set to M1 and A1 before "looking inside" the circuit.

1.2 Summary of Contributions

The learning methods proposed in this thesis all try to extract useful lessons from single examples. Those lessons are then used to recognize similarities between new examples and previous examples that have already been solved. We ask what kinds of similarities can be exploited and what lessons can be extracted that will enable recognition of those similarities. Thus, the overall theme is to extract as much useful information as possible from single diagnostic examples.

There are three main contributions of this research:

- We present many notions of similarity for diagnostic examples. Trying to recognize each type of similarity provides an interesting possibility for learning.
- We analyze the effect on performance of looking for each kind of similarity.

- We analyze the strengths and weaknesses of Explanation-Based Generalization (EBG), a technology that is used in several learning methods throughout the thesis.

1.3 Multiple Definitions of Similarity

Each different type of similarity suggests a different set of lessons that a learning program should extract from examples. One contribution of this research is to propose several definitions of similarity for diagnostic cases and learning components that generalize examples based on each of the definitions. The definitions of similarity are summarized below.

Chapter 2 discusses classifying two cases as similar if the same pattern of inferences applies to both. For example, as already mentioned, in Figure 1.1 the two sets of observations for the device are similar because the observations can be propagated through the same components, in the same order, leading to a contradiction at the same location. That is, the same pattern of inferences is applicable to both sets of observations.

We can relax that definition of similarity for sets of device observations if we do not require that the sequence of value propagations use exactly the same components, but only components that play the same role in the circuit. Section 6.1.1 defines similarity for patterns of inferences in terms of equivalent roles played by components. For example, propagating a value through the first bit-slice of a carry-chain adder is similar to propagating a value through the second bit-slice. As a result, two sets of observations for a carry-chain adder can be defined as similar if "similar" patterns of inferences are applicable to them: that is, if propagating values through either the first or the second bit-slice leads to a contradiction.

Chapter 5 discusses classifying two sets of observations as similar if they can be caused by the same misbehavior of a particular component. For example, the two sets of observations in Figure 1.1 can both be caused by the first bit of M1's output being stuck-at 0.

Again, we can relax that definition of similarity for sets of device observations if we do not require exactly the same component misbehavior to explain the two sets of observations, but only a similar misbehavior. Thus, Section 6.2.1 defines two misbehaviors for different components as similar if the components play equivalent roles in the circuit and the misbehavior is the same. For example, in Figure 1.1, the first bit of M1's output being stuck-at 1 is similar to the first bit of M3's output being stuck-at 1. Section 6.2.2 proposes a different definition of similarity for component misbehaviors. There, the first bit of M1's output being stuck-at 1 is defined as similar to the first bit of M1's output being stuck-at 0. Using either of those notions of similarity for component misbehaviors, we define two sets of observations as similar if they can both be explained by "similar" component misbehaviors.

In order to recognize each of the types of similarity described above, a learning system has to extract appropriate lessons from the examples it solves. The programs described in this thesis use explanation-based methods to construct those lessons (which we call generalized rules). Explanation-based methods have two advantages over statistical similarity-based methods. First, explanation-based methods use domain knowledge to guide generalization from single cases, while statistical methods require numerous cases. This allows the explanation-based learner to learn more quickly, based on fewer cases. Second and more important, in contrast to statistical similarity-based methods, the explanation-based methods we use do not require an inductive bias as to the correct language and form in which to describe classes of cases. Instead, the language used to describe the behavior and misbehavior of components provides the language to use in classifying device misbehavior.

1.4 Selecting Useful Definitions of Similarity

Some notions of similarity are more useful than others for speeding up performance, because there is a cost to looking for similarities. If the benefits gained from exploiting similarities in solving new examples do not outweigh the costs of looking for the similarities, performance will even deteriorate. Each notion of similarity (e.g., same pattern of inferences) gives rise to several generalized rules. Some insight into which notions are useful in improving performance can be gained from analyzing which individual generalized rules will improve performance. We gain even stronger insights by analyzing the aggregate effects of all of the generalized rules.

1.4.1 Utility of an Individual Generalized Rule

While we assume that the fixed cost of *constructing* a generalized rule will be amortized over an indefinite number of cases, and hence can be ignored, it is wise to examine the costs and benefits of *using* a generalized rule. We suggest three criteria that a generalized rule must satisfy in order to improve the performance of a problem solver:[1]

Recurrent Not only must a generalized rule apply to many cases (the traditional generality criterion), it must also apply to the cases that the problem-solver will actually encounter.

Manifest It must be inexpensive to check whether a generalized rule applies to a new case.

Exploitable The knowledge that the generalized rule applies to a new case must provide some discriminatory power in reasoning about the case.

[1]These three factors were also identified independently in [Min88, MCE+87].

Of course, these criteria are not binary predicates: a generalized rule may be more or less recurrent, manifest, or exploitable. Consider for a moment why rules that do not satisfy these criteria will *not* be useful in speeding up performance. If a generalized rule's applicability can be checked at almost no cost (manifest) and provides a complete solution to the cases in which it is applicable (exploitable), but it is not applicable to *any* of the cases that the system is presented with (no recurrence), checking that generalization will slow down the system. Similarly, if a generalization is applicable in nearly every case, and knowing it is applicable provides a complete solution, but it costs more to check the generalized rule than to solve the problem from first principles, again, checking the generalized rule only slows down the system. Finally, if a generalized rule applies to nearly all of the cases and can be recognized at almost no cost, but it provides no discriminatory power in reasoning about the case (e.g., a rule that applies to every set of observations), there is no advantage to using it.

The utility of a notion of similarity is just the sum of the utilities of the generalized rules it gives rise to. Hence, there are three analogous criteria for evaluating the utility of a notion of similarity. First, large numbers of cases that the troubleshooting engine is likely to encounter should be similar according to that notion of similarity. Second, it should be inexpensive to recognize that kind of similarity. Finally, recognizing that the current case is similar to a previous case should help in solving the current case.

1.4.2 Aggregate Utility

We can gain further insight into the utility of a notion of similarity by analyzing the aggregate effects of all of the generalized rules. Chapter 3 presents such an aggregate analysis for the learning system described in Chapter 2. That learning system defines two sets of observations as similar if the same derivation of contradictory values is applicable to both. Experimental results show that single-fault diagnostic speed improved on both the circuit shown in Figure 1.1 and a gate-level implementation of a carry-lookahead adder. More importantly, we present a breakdown of the operations involved in diagnosis, both with and without the generalized rules. The cost breakdown enables rough predictions about how the learning system will affect performance on other devices. The cost breakdown also enables predictions about how changes to the original diagnostic engine would affect the utility of the learning system.

1.5 Performance Effects of EBG

Chapter 4 takes a closer look at Explanation-Based Generalization, or EBG [MKKC86], a technology used in the learning system of Chapter 2 and several of the learning systems described in Chapters 5 and 6. Again, we give an aggregate analysis, viewing the generalized rules constructed by EBG in terms of changes to the search strategy of a problem solver. We analyze the sources of power in two common uses of EBG to improve performance: generalizing successful problem solving episodes, and generalizing the explanations that search nodes are inconsistent (often referred to as learning from failure [Min88, MB87, Ham87, Paz86]). We summarize that analysis below.

1.5.1 Generalizing Successful Patterns of Inferences

There are two sources of power in using EBG to generalize successful problem solving episodes. First, the generalized rules can bias the problem solver's search toward patterns that have been useful in solving previous problems, hence away from patterns that have never been useful. Second, the generalized rules *encapsulate* patterns of operator applications: the program can check the preconditions of a whole pattern and jump to the conclusions without ever incurring the overhead costs of binding variables for the operators and storing intermediate results. The following highlights some key observations from our analysis:

Biasing Search EBG on its own does not capture enough information about the frequency with which patterns of inferences are applicable to be able to bias search in the most effective way possible. EBG is a technique for learning from a single example, but frequencies of applicability are properties of whole distributions of examples.

Encapsulation The benefits from encapsulation depend on the relative cost of evaluating the bodies of search operators versus the cost of binding variables for the operators and storing the results of operator applications.

Caveat: Searching For All Solutions If the problem solver's task is to find *all* of the solution states, using EBG to identify one or a few solution states may not reduce the search for the rest of the solution states.

1.5.2 Generalizing Proofs of Inconsistency

There are two potential sources of power in using EBG to generalize explanations of the inconsistency of search nodes. First, proving the inconsistency of a search node may be very expensive; recognizing that a previously successful derivation of an inconsistency is applicable may reduce that cost. In this case, generalizing explanations of failures is the same as generalizing *successful* patterns of inferences in the space of derivations of inconsistencies. Performance may improve due to either search reduction or encapsulation, or both.

Second, knowing the inconsistency of some search nodes may enable the problem solver to ignore a large portion of the original search space. The problem solver may cut off search either below or above the search nodes that the generalized rules identify as inconsistent.

Cutting Off Below Inconsistent Nodes If goal nodes are never reached from inconsistent nodes, the problem solver can cut off search at a node that a generalized rule identifies as inconsistent. One must be careful in measuring these gains, however, because a well-designed original problem solver may be able to cut off search below inconsistent nodes even without the generalized rules.

Cutting Off Above Inconsistent Nodes The problem solver may be able to combine information provided by more than one generalized rule to cut off search above the nodes that the generalized rules identify as inconsistent. One example of this is the use of the single-fault assumption in diagnosis to intersect sets of components that the generalized rules identify as inconsistent.

1.6 Map of the Thesis

In summary, Chapters 2, 5, and 6 present several ways that diagnostic examples can be thought of as similar, and how single examples can be generalized in order to capture those similarities. Since there are costs as well as benefits to using generalizations, performance analysis permeates the entire thesis. Chapter 3 in particular presents a detailed performance analysis and experimental results for the learning program described in Chapter 2. Chapter 4 analyzes the sources of power in Explanation-Based Generalization.

Chapter 2

Similar = Same Contradiction Derivations

This chapter describes a learning system that remembers useful patterns of inferences and checks their applicability to new diagnostic cases. That is, two sets of observations for a given circuit are considered similar if certain patterns of inferences are applicable to both. The particular patterns that are of interest are those that derive contradictions by propagating *values through the circuit components.* The original troubleshooting engine uses derivations of contradictory values to identify "conflict sets," sets of components that cannot all be working properly. The learning mechanism creates a generalized rule from each derivation. The generalized rules are then used to check the applicability of the derivations to future sets of observations for the same circuit. The generalized rules can speed up diagnosis by identifying conflict sets faster than the original diagnostic engine can identify them through value propagation.

Section 2.1 presents a single-fault candidate generator for model-based diagnostic problems.[1] Section 2.2 presents the learning mechanism, which uses Explanation-Based Generalization to encapsulate derivations of contradictory values. Section 2.3 describes an augmented diagnostic engine that uses the learning mechanism to construct generalized rules, and then uses the generalized rules in diagnosing future cases. Diagnosis of the polybox circuit is used throughout to illustrate the algorithms. Section 2.4 provides an extended example that demonstrates how the system constructs and uses generalized rules in diagnosis of a gate-level implementation of a carry-lookahead adder. Chapter 3 presents experimental results that demonstrate that the augmented diagnostic algorithm can improve performance.

[1]Section 4.2.2 discusses why the technique described here would not speed up the multiple-fault candidate generation process used in GDE [dKW87].

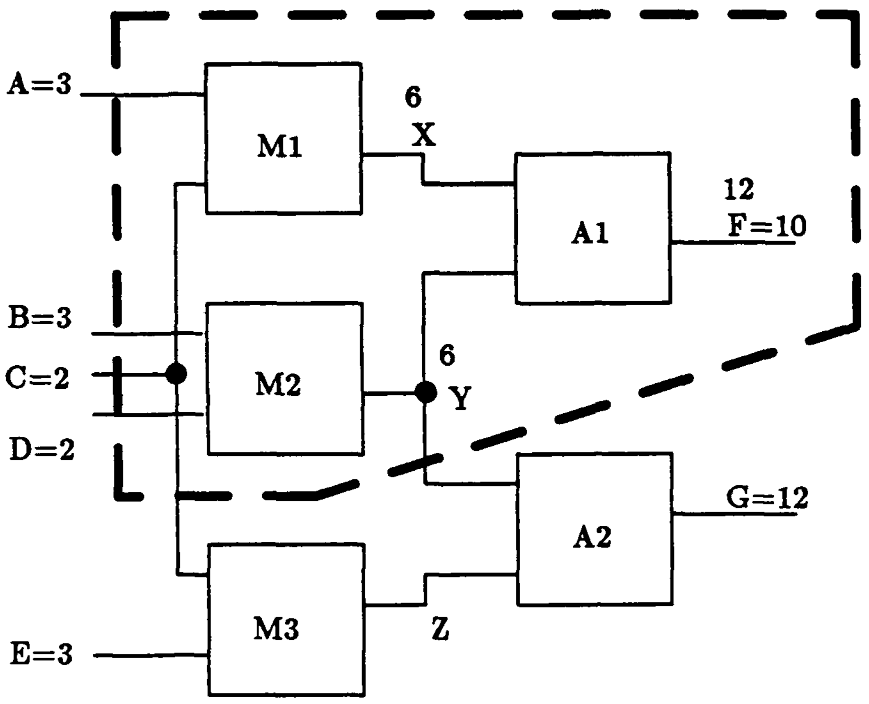


Figure 2.1: The first case

2.1 Candidate Generation

This section presents the single-fault candidate generation method described in [HD87, Dav84, Gen84]. Device structure is described by a list of components and their interconnections. Component behavior is modeled by rules for inferring an input or output from other inputs and outputs. For example, the behavior of an adder is modeled by three rules. The first computes the output by adding the two inputs. The other two each compute one of the inputs by subtracting the other input from the output. Thus, given values on any two "ports" of the adder, the rules predict a value on the remaining port.

Given observations of the values at the inputs and outputs of a circuit, the candidate generation program first propagates input values through the circuit, using the circuit structure and the component behavior rules. For example, in Figure 2.1, the behavior rule for multiplier M1 predicts 6 at X from the inputs 3 and 2. If the circuit is malfunctioning, predicted values at the outputs will contradict the values observed, as in Figure 2.1, where the predicted value of 12 at F contradicts the observed value of 10.

The next question the program asks is: which components were used in determining the predicted value at the contradiction site? The program calculates that set, called a *conflict set*, by tracing back through a dependency trail to find those components whose behavior rules were used in predicting the contradicted value.[2] For instance, in the first example above, `(M1 M2 A1)` is a conflict set, since M1, M2, and A1 are the only components that are needed to predict the value 12 at F.

Conflict sets are valuable because they restrict the troubleshooter's attention to the components that can account for the observed symptoms. If component M3 is not in a conflict set (e.g. `(M1 M2 A1)`), its behavior is irrelevant to the associated contradiction: the contradiction will exist no matter how the component is behaving. M3 may or may not be working, but it cannot explain the contradiction at F. [3]

The candidate generator described here looks for all of the single point of failure candidates. If a single component is to account for all of the observed misbehavior, it must be in every conflict set. Hence, the troubleshooting engine keeps track of the intersection of all the conflict sets found so far, which we term the *suspect* set. In other words, any component in the complement of a conflict set is exonerated.

Each suspect is then tested by a process called constraint suspension [Dav84]: the program assumes that all of the other components are working but disables the suspect's behavior rules (*suspends the constraints* it places on circuit values). If the remaining components can be used to derive a contradiction, the suspect is ruled out, since it cannot account for that symptom. In addition the program identifies another conflict set, possibly further restricting the suspect set. If no contradiction is derived, the suspect is a candidate.

In the example of Figure 2.2, the initial suspect set is `(M1 M2 A1)`. Constraint suspension is performed on M2 (Figure 2.2). Disabling the behavior rules for M2 resolves the contradiction at F by making it impossible to predict the value 12 there. Another contradiction is predicted, however, at Y: A1 predicts the value 4 (from 10 at F and 6 at X), while A2 predicts 6 (from 12 at G and 6 at Z). The new conflict set is `(A1 A2 M1 M3)`. One of the components in each conflict set must be broken. Thus, by the single-fault assumption, the broken component must be in the intersection of the conflict sets. Intersecting reduces the suspect set to `(M1 A1)`. In this case, only the component which was suspended, M2, is exonerated. If a contradiction is found during constraint suspension, the suspended component will always be exonerated. In general, other components may be exonerated as well when the new conflict set is intersected with the previous ones.

The remaining suspects, M1 and A1, are then tested in turn, but no further contradictions are found. M1 and A1 are the single-fault candidates.

Candidate generation is a winnowing process: suspect components that cannot

[2]Our troubleshooter propagates values and recovers the conflict sets by tracing dependency records, unlike the ATMS-style diagnosis of [dKW87] that propagates environments and thus builds conflict sets as part of the propagation process. Section 3.8 discusses the performance effects from learning that would occur using an ATMS implementation of single-fault candidate generation.

[3]This assumes that the model we are given is correct; in particular that its topology correctly models the connectivity of the circuit. To see what happens when this is not true, see [Dav84].

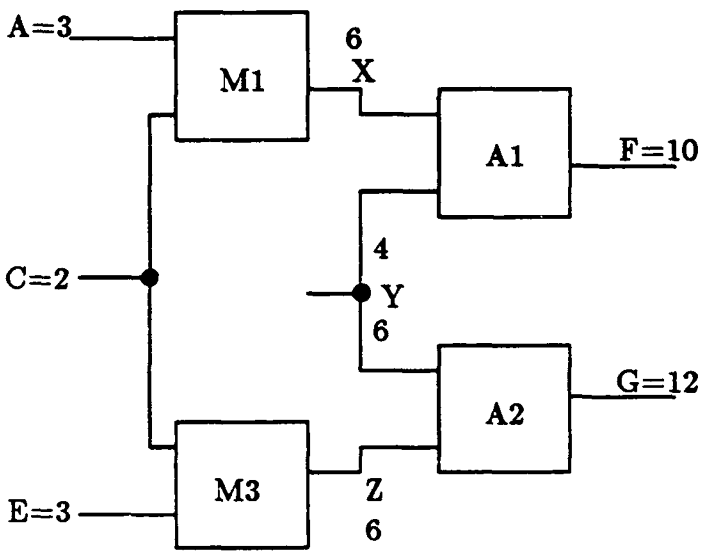


Figure 2.2: Constraint Suspension on M2

1. Assume all components are working. Propagate values from inputs to outputs to find an initial contradiction, yielding an initial suspect set.
2. While there are still unexamined suspects:
   - (a) Choose an unexamined suspect at random.
   - (b) Perform constraint suspension on the suspect.
   - (c) If a contradiction is found, form conflict set, which reduces suspect set.
   - (d) If no contradiction is found, add suspect to candidate set.
   - (e) Remove suspect from unexamined suspect set.

Figure 2.3: The Original Diagnostic Algorithm

account for all of the misbehavior of the device are exonerated. An effective way to determine the candidates is to identify the conflict sets. Hence, any process that can speed up the identification of conflict sets has the potential to speed up the system's performance.

2.2 The Learning Component

Now suppose that the program is used repeatedly to diagnose devices with the same design description, but is given different sets of observations each time. Almost any hardware device seems to have a few weak links that break more frequently than the rest of the components of the device. Hence, it is likely that the diagnostic engine repeatedly will perform the same sequences of value propagations that predict contradictions. That is, the program may be given different observations, but will frequently predict contradictory values by propagating those observations through the same sequence of components.

This section describes an algorithm that creates a generalized rule from a derivation of contradictory values. Once created, the generalized rule can check efficiently whether all of the steps of a derivation apply to a new case, without actually performing the derivation. When the derivation does apply to a new case, the troubleshooter will use the rule to jump to the conclusion, and construct the conflict set without propagating values through the circuit.

2.2.1 Explanation-Based Generalization

Explanation-Based Generalization (EBG) [MKKC86] is a widely known method of using domain knowledge to learn from a single example. EBG constructs sufficient conditions for concluding that a pattern of inferences is applicable. This section presents the EBG framework and describes how our generalization machinery maps onto it. Readers unfamiliar with the EBG framework may choose to skip to the examples in Section 2.2.3 before reading this section.

In EBG, the system is given a goal-concept (a description of a class of examples), a positive instance of that concept, a domain theory, and an operationality criterion. The initial formulation of the goal concept does not satisfy the operationality criterion. The task is to find a reformulation that does satisfy the operationality criterion, using the training example and the domain theory. The performance program uses the domain theory (a set of inference rules) to prove (explain) that the training example satisfies the given formulation of the goal concept. A generalization algorithm then finds the weakest set of preconditions under which the same proof would apply. These preconditions ignore "irrelevant" features, and replace constants with predicates on variables. If these preconditions satisfy the operationality criterion, they are the desired reformulation of the goal concept.

Because a single explanation of why the instance satisfies the goal concept guides the generalization, the reformulation is a specialization of the original goal concept. It gives necessary and sufficient conditions for the particular explanation to be applicable. If other explanations are possible, however, this reformulation provides only *sufficient* conditions, and not necessary conditions, for recognizing future cases as instances of the goal concept.

2.2.2 EBG on Conflict Set Derivations

In generalizing from a candidate generation case, there will be one goal concept (and hence one generalized rule) that corresponds to each conflict set found in diagnosing the case.

Goal Concept The sets of observable values for the device from which a given set of components can predict contradictory values. For example: "The sets of observations for which `(M1 M2 A1)` is a conflict set."

Training Example The training example consists of one set of observations for the device that is a positive instance of the goal concept. For example, (A=3; B=3; C=2; D=2; E=3; F=10; G=12) is a set of observations for which `(M1 M2 A1)` is a conflict set.

Domain Theory The domain theory consists of the structure of the device and the behavior descriptions of the device's components.

Operationality Criterion Since the purpose of generalizing is to enable the program to make some diagnostic inferences before tracing through the structure of a circuit, the operationality criterion requires that predicates be testable on sets of observations without propagating values in the circuit. The reformulated goal concept can thus mention only the observables of the circuit, and not internal values.

Proof The dependency trail of the derivation of contradictory values serves as the proof that the training example satisfies the goal concept.

Our algorithm for finding the weakest preconditions replaces the actual observations with variables, and works forward through the proof tree, running each of the behavior rules to predict symbolic values (i.e., expressed in terms of variables).[4] Behavior rule firings that occur later in the derivation then propagate the symbolic values derived. Some restrictions on the symbolic values may be needed to satisfy the preconditions of the behavior rules. These restrictions, together with a predicate which ensures that the two values derived are indeed contradictory, form the preconditions of the generalized rule.

2.2.3 Example: Generalizing the Derivation of Conflict Set (M1 M2 A1)

[4]Our algorithm bears a close resemblance to that of [DM86], which corrects a technical error in the algorithm presented in [MKKC86].

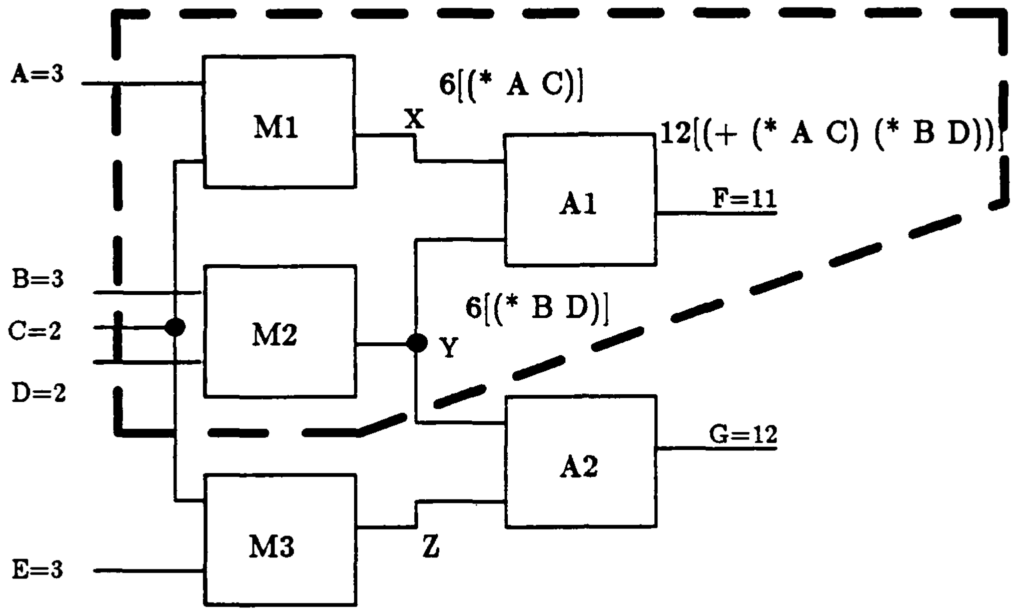


Figure 2.4: Generalizing the construction of conflict set (M1 M2 A1). Symbolic values propagated are in brackets.

Figure 2.4 illustrates how the program generalizes from the derivation of the conflict set (M1 M2 A1). It first substitutes variables A, C, B, D, and F for their respective values, then reruns the behavior rules. M1's forward behavior rule predicted 6 at X; the symbolic value predicted is (* A C), as shown in brackets in Figure 2.4. Similarly, M2 predicts (* B D) at Y. A1 uses those values to predict (+ (* A C) (* B D)) at output F. A contradiction will arise whenever the observed value at F differs from the value of this expression. The resulting rule in this case is:[5]

```
R1: IF (NOT (= ?F (+ (* ?A ?C) (* ?B ?D))))
    THEN (CONFLICT-SET '(M1 M2 A1))
```

What generalization can we make of the construction of the second conflict set (M1 M3 A1 A2), which exonerated M2? A common answer is the rule that "if F or G is incorrect but the other is not, then M2 cannot be a suspect."[6] The intuition is that if M2 is broken, there should be incorrect outputs at both F and G, not just at one of them. If they are both incorrect, the intuition is that M2 can be a candidate, because it contributes to both outputs.

The program creates a rule that states:

[5]Throughout this thesis, symbols preceded by question marks indicate variables which must be bound before the rule can be fired. When R1 is checked in a new case, ?F will be bound to the observed value at output F, ?A to the observed value at input A, and so on.

[6]Stating that (M1 M3 A1 A2) is a conflict set is equivalent to stating that M2 is not a suspect.

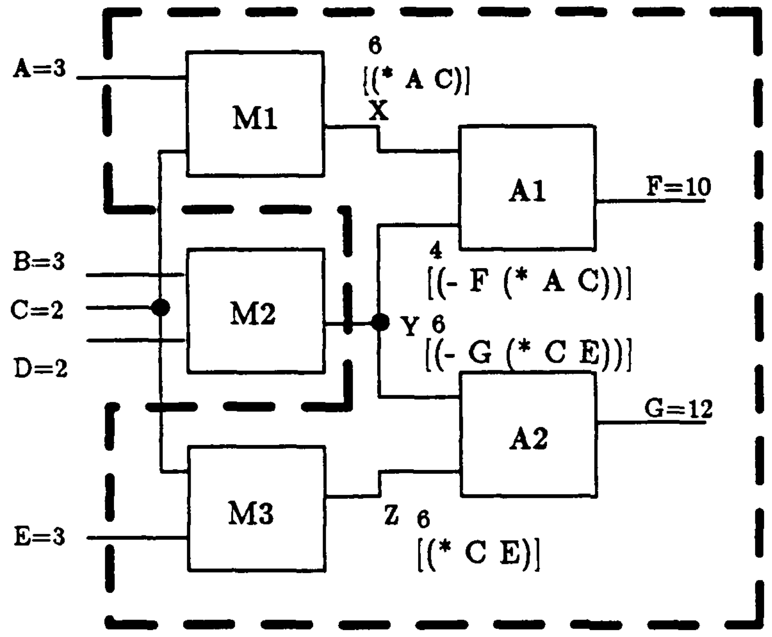


Figure 2.5: Generalizing the construction of conflict set (M1 M3 A1 A2)

```
R2: IF (NOT (= (- ?F (* ?A ?C))
               (- ?G (* ?C ?E))))
    THEN (CONFLICT-SET '(M1 M3 A1 A2))
```

This rule applies when only one of the two outputs is incorrect, but it also applies in many cases where *both* outputs are incorrect. For example, R2 applies when the observables are (A=3; B=3; C=2; D=2; E=3; F=10; G=8), which is a case not covered by the generalization produced by common intuition. Common intuition fails because even though M2 can account for the misbehavior at *either* F or G, it would have to be malfunctioning in two different ways, producing the outputs 4 and 2, in order to explain the misbehavior at *both* F and G. Thus, the computer-generated rule correctly exonerates M2 given these observations, whereas the rule produced by common intuition does not.

2.3 The Augmented Diagnostic Algorithm

We implemented an augmented diagnostic engine that uses the generalized rules created by EBG to improve diagnostic performance on cases that are "similar" to cases the program has diagnosed before. The generalized rules enable the program to recognize when a pattern of inferences from a previous case can be applied to a new case, and to jump to the same conclusion, the identification of a conflict set. The diagnostic program starts with a reduced suspect set if the generalized rules identify

1. Retrieve from the library for the device the generalized rules for noticing conflict sets. Check the applicability of each rule and intersect the conflict sets identified to form the initial suspect set.
2. If there are no conflict sets found using generalized rules, propagate values from inputs to outputs to find an initial contradiction, yielding an initial suspect set.
3. While there are still unexamined suspects:
   (a) Choose an unexamined suspect at random.
   (b) Perform constraint suspension on the suspect.
   (c) If a contradiction is found, form a conflict set, which reduces the suspect set.
   (d) If no contradiction is found, add the suspect to the candidate set.
   (e) Remove the suspect from the unexamined suspect set.
4. Use EBG to generalize each derivation of contradictory values found by propagating values and add the new rules to the library.

Figure 2.6: The Augmented Diagnostic Algorithm

some conflict sets, which reduces the total number of suspects on which constraint suspension must be performed. The program then falls back on constraint suspension to try to exonerate the remaining suspects. Figure 2.6 gives a more precise description of the augmented diagnostic algorithm.

The augmented diagnostic engine must fall back on constraint suspension after using its past experience because it can never be sure if it has a complete set of generalized rules. There might be some derivations of contradictory values that the program has *not yet* encountered, in which case the generalized rules might miss identifying some conflict sets. In order to guarantee that the augmented diagnostic engine exonerates all of the components that it is possible to exonerate, the program performs constraint suspension on each of the initial suspects.

Note that falling back on constraint suspension of the initial suspects places a lower bound on how fast the augmented diagnostic engine can run. No matter how good the generalized rules are, the augmented diagnostic engine will perform constraint suspension on *at least* each of the final candidates. The generalized rules can only be used to save the cost of performing constraint suspension on some components that are eventually exonerated.

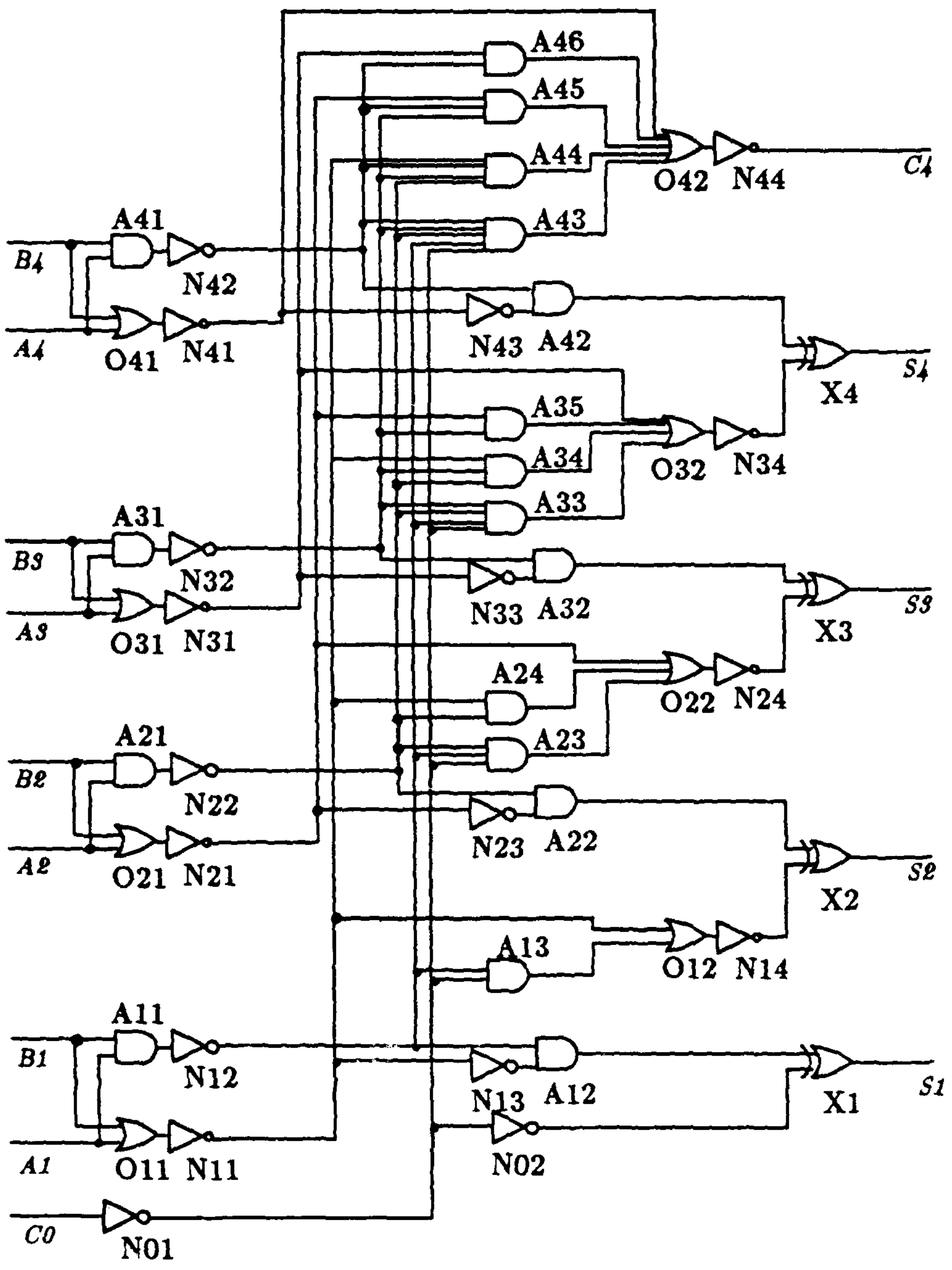


Figure 2.7: The gate-level description of a carry-lookahead adder, adapted from the TTL Data Book.

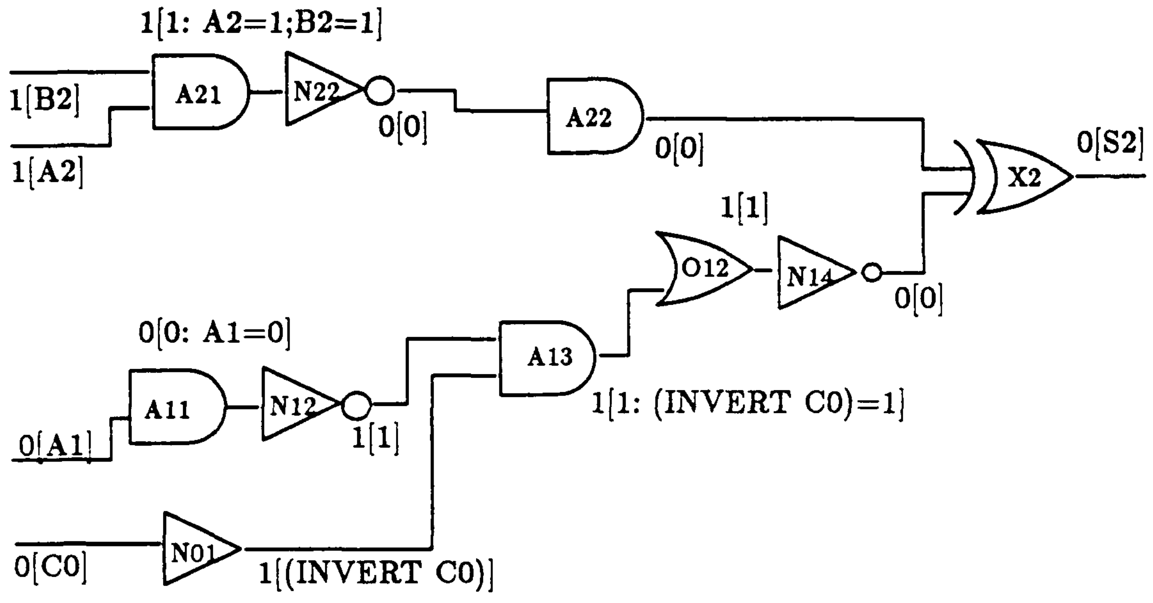


Figure 2.8: The derivation and generalization of a contradiction that yields the conflict set (A21 N22 A22 A11 N12 N01 A13 O12 N14 X2). Generalized values predicted are in brackets, followed by preconditions, if any, for the rule firing.

2.4 The System in Action: A Carry-lookahead Adder

We use the carry-lookahead adder in Figure 2.7 to illustrate the augmented algorithm in action. As we will see, the program first diagnoses an adder that produces the output 10 from inputs 6 and 6. It then constructs two conflict sets from two derivations of contradictory values in the circuit. It creates two generalized rules from those two derivations, and inserts the two rules into the library. In diagnosing another adder, which produces the output 17 from inputs 7 and 14, one of the rules generated in diagnosing the first case applies, but the other does not, and the program falls back on constraint suspension to find an additional conflict set.

2.4.1 The First Case: 6 + 6 = 10

The first case presented to the system has inputs 6 and 6 (carry-in C0 is 0), and output 10. There are no generalized rules in the library yet, so the program proceeds to step two of the augmented diagnostic algorithm. The conflict set found in Figure 2.8 is (A21 N22 A22 A11 N12 N01 A13 O12 N14 X2). Special-case behavior rules for and-gates and or-gates can predict the gate's output from just one input in the obvious situations, as for example A22 in Figure 2.8. Using the special-case rule,

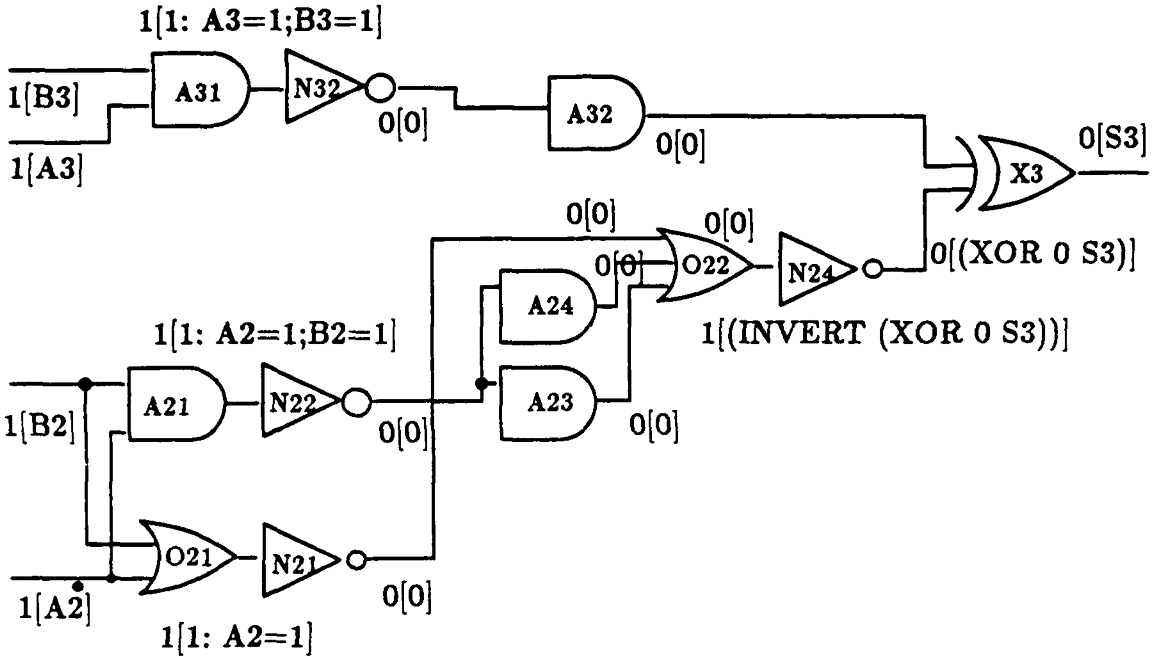


Figure 2.9: The derivation and generalization of a contradiction that yields the conflict set (A31 N32 A32 X3 N24 O21 N21 N22 A24 A23 O22 A21). Generalized values predicted are in brackets, followed by preconditions, if any, for the rule firing.

A22's output depends on fewer components, so a smaller conflict set can be created.

The program performs constraint suspension on X2 in step three, and another contradiction is derived (Figure 2.9). The conflict set identified is (A31 N32 A32 X3 N24 O21 N21 N22 A24 A23 O22 A21). The diagnostic engine intersects the two conflict sets, which reduces the suspect set to (A21 N22). Constraint suspension is performed on A21 and N22 in turn, but no further conflict sets are found.

The program then creates two rules for recognizing the applicability of the two derivations of contradictory values. The figures illustrate this generalization process. Note that some restrictions on the symbolic values are needed in order to satisfy the preconditions of the behavior rules. For example, in Figure 2.8, the behavior rule for A13 that produced output 1 required both of its inputs to be 1. The symbolic values on A13's inputs are 1 and (INVERT CO), so when the behavior rule is run during the generalization process, output 1 is predicted and the precondition (= 1 (INVERT ?CO)) is added to the generalized rule's preconditions. The two rules generated are:

```
R4:
 IF (AND (NOT (= 0 ?S2))
         (= 1 (INVERT ?CO))
         (= 0 ?A1)
         (= 1 ?A2)
         (= 1 ?B2))
 THEN (CONFLICT-SET '(A21 N22 A22 A11 N12 N01 A13 O12 N14 X2))

R5:
 IF (AND (NOT (= (INVERT (XOR ?S3 0)) 0))
         (= 1 ?B2)
         (= 1 ?A2)
         (= 1 ?A3)
         (= 1 ?B3))
 THEN (CONFLICT-SET '(A31 N32 A32 X3 N24 O21 N21 N22 A24 A23 O22 A21))
```

2.4.2 The Second Case: 7 + 14 = 17

Imagine that the augmented diagnostic program is later given another copy of the adder circuit to diagnose, with the observables A=7, B=14, C0=0, S=17. Here R5 applies; R4 does not apply because input A1 is 1, not 0 as it requires. Hence, the initial suspect set is (A31 N32 A32 X3 N24 O21 N21 N22 A24 A23 O22 A21). When constraint suspension of A21 is performed, a contradiction is found, yielding the conflict set (O21 N21 O11 N11 A24 N23 A11 N12 N01 A13 O12 N14 X2 A22 A23 O22 N24 X3 A32 N32 A31). This reduces the suspect set to (A31 N32 A32 X3 N24 O21 N21 A24 A23 O22). Since no further contradictions are found, this is the final candidate set.

The final candidate set (A31 N32 A32 X3 N24 O21 N21 A24 A23 O22) is the same one that would have been produced without using the generalized rules. The augmented diagnostic program, however, finds one of the conflict sets faster than the original diagnostic program would have, because of the applicability of R5, which was created during diagnosis of the first case.

2.4.3 Summary

The diagnosis of the two cases for the adder circuit illustrates three important points about the augmented diagnostic algorithm. First, a generalized rule constructed from a single case was applicable to a second case that bears little surface resemblance to the initial case. Second, while two rules were generated in diagnosing the first case, only one of them was applicable in diagnosing the second. Cases can be "similar" according to one generalized rule, but not "similar" according to another. This is not surprising since each generalized rule defines similarity by the common applicability of a particular pattern of inferences. Third, the generalized rules allow the program to start with a small initial suspect set, but there might be some derivations of contradictory values that the program has not yet encountered, so that some conflict sets are not identified using the generalized rules. Hence, the program performs constraint suspension on each initial suspect to try to further reduce the suspect set.

Chapter 3

Performance Analysis

The previous chapter illustrated how a diagnostic program can benefit from recognizing that derivations of conflict sets from earlier examples are applicable to later examples. It is important to realize, however, that in order to gain those benefits, the program pays the cost of checking all of the generalized rules. Throughout this chapter, the term 'utility' refers to the difference between the benefits and the costs.

There is no a priori reason to expect the benefits of the generalized rules to outweigh the costs, or vice-versa. In this chapter, we present experimental results measuring the utility of the generalized rules that are constructed during diagnosis of the polybox and adder circuits. The experiments measure only the effect of *using* the generalized rules, under the assumption that the cost of creating the generalized rules will be amortized over enough cases so as to be negligible. Performance on both the polybox and the adder circuit improved using the generalized rules. Hence, at least for the selection of cases used in the experiments, the benefits of the generalized rules outweighed the costs.

We also present a breakdown of the operations used in diagnosis, with and without generalized rules. This culminates in Section 3.4.3 with an equation for the change in performance caused by the generalized rules. In Section 3.7, the equation is used to sketch the device characteristics that influence the utility of the generalized rules. In Section 3.8, the equation is used to predict how changes to the diagnostic engine would affect the utility of the generalized rules.

3.1 Experiment Description

We ran experiments to compare the efficiency of the original diagnostic program to the efficiency of the augmented diagnostic program on the polybox and adder circuits. First, a large space of cases was generated for each circuit. Each experiment started with the random selection of a set of training cases and a set of test cases from that space. Then, statistics were gathered for each of the following runs:

1. Each training case is diagnosed using the original diagnostic program (i.e. without generalized rules.) This establishes a baseline for comparison.

2. Each training case is diagnosed again. This time, the program creates a new generalized rule each time it finds a new way of deriving contradictory values. Thus, in diagnosing the tenth case, the program used the generalized rules it created during diagnosis of the previous nine. The results from the second run are compared with the results from the first run in order to measure the effect of "continuous" learning.

3. The third run establishes a baseline for the test cases. Each test case is diagnosed without creating or using any generalized rules.

4. Each test case is then diagnosed again, using the library of generalized rules created in the second run. No new generalized rules are constructed. The results from the third and fourth runs are compared in order to measure performance "after" learning.

Note that both the second and fourth runs provide a way of measuring the transfer of applicability of the generalized rules to new cases. For both the polybox and adder circuits, performance improved during continuous learning and was better after learning than before.

3.2 Performance Results

3.2.1 Polybox Experiments

Cases Generated

In order to generate cases (sets of observations) for the polybox circuit, we selected 79 sets of inputs at random.[1] Then, for each set of inputs we generated the outputs that would be produced by each of of the possible stuck-at faults for the components. A stuck-at high at a component's port models a wire as always carrying the value 1, even when it should be carrying the value 0. This type of error occurs frequently because the connections between pins and the internals of a chip come loose. Similarly, stuck-at low indicates that the wire always carries the value 0. Each of the 148 possible stuck-at faults on the inputs or outputs of the circuit components were considered. For example, multiplier M1 has two three-bit inputs and a six-bit output. Each of these can be stuck either high or low, so there are a total of 24 possible stuck-at faults for M1. Similarly, adder A1 has two six-bit inputs and a seven-bit output, each of which can be stuck either high or low, for a total

[1]There is no significance to this number. There were 2^{15} possible sets of inputs. 79 sets of inputs yielded 4376 input/output combinations, which seemed like enough.

of 38 possible stuck-at faults. The circuit was simulated using each possible fault model (assuming that the rest of the components were working properly) to predict the output values.

Results

We randomly selected 100 training cases and 100 test cases from among those generated. The average time taken to diagnose a training case without using any generalized rules was 0.77 seconds. The average time to diagnose a training case dipped to 0.56 seconds when generalized rules from previous cases were used. Only three generalized rules were constructed during this second run, because there are only three ways to derive conflict sets in the polybox circuit. The average time to diagnose a test case without using any generalized rules was .76 seconds. That dipped to .55 seconds when using the three generalized rules created from the training cases. Thus, the three generalized rules improved performance on the polybox circuit by 28%.

3.2.2 Adder Experiments

Cases Generated

The space of adder cases was generated by considering all of the possible input/output combinations in which the carry-in bit C0 was 0.[2] These were then filtered to keep only those cases that admitted single-fault candidates. A total of 1092 cases passed the filter.

Results

Performance improved in diagnosing the adder circuit as well. We randomly selected 150 training cases and 150 test cases from among those generated. The average time taken to diagnose a training case without using any generalized rules was 7.09 seconds. The average time to diagnose a training case dipped to 6.23 seconds when generalized rules from previous cases were used. A total of 221 generalized rules were constructed during that run. The average time to diagnose a test case without using any generalized rules was 7.07 seconds. That dipped to 5.66 seconds when using the 221 generalized rules created from the training cases. On average, 3.15 generalized rules applied to each test case and 0.56 additional conflict sets were found using constraint suspension. Of the 5.66 seconds, 0.70 seconds was spent checking

[2]The restriction that C0 be 0 is physically plausible because the carry-in bit is not needed in some uses of an adder, in which case the pin is tied to ground. For these experiments, the restriction is more pragmatic than principled: it took several days to generate the cases even with the restriction.

the generalized rules. Overall, the generalized rules improved performance on the adder circuit by 20%.

3.3 Case Selection for Experiments

Unfortunately, the correct way to benchmark a performance learning system is still an open question. One difficult issue is the distribution of training and test cases. We sampled cases uniformly from among those generated, but any practical circuit would fail in some ways more frequently than in other ways. By paying more careful attention to how we generated cases (e.g. implementing the circuits with TTL chips and then simulating the typical ways that TTL chips fail) it would have been possible to produce a distribution of cases that was arguably more realistic. We do not claim to have chosen the "correct" distribution of examples. Hence, the experimental results only serve to illustrate that using EBG to generalize derivations of conflict sets can improve performance and to motivate the analysis of the factors affecting whether it will do so.

Another issue is how robust the results are. What if smaller or larger training and test sets were selected? What if different random samples of the same size were selected? Appendix B reports experiments showing that, with either of those changes to the training and test sets, performance still improves.

3.4 Cost Breakdown

This section highlights the costs of the different operations involved in diagnosis, both with and without the learning component. The resultant cost formula is then used to identify the sources of the speedup in the experiments. Section 3.7 uses the formula to give qualitative characteristics of the circuits for which EBG will improve performance and Section 3.8 uses the cost breakdown to argue that changes in the relative efficiency of the operations that are used by the original diagnostic program can alter the direction and the magnitude of the performance change resulting from using EBG.

3.4.1 Without Generalized Rules

The costs of running the original diagnostic program can be split into three categories. First there are the costs of propagating values. Second, there are the costs of switching assumptions about which components are working, which happens during constraint suspension. Finally, there are the costs of constructing conflict sets when contradictory values are found. This section analyzes these three costs in turn.

1. Assume all components are working. Propagate values from inputs to outputs to find an initial contradiction, yielding an initial suspect set.
2. While there are still unexamined suspects:
    (a) Choose an unexamined suspect at random.
    (b) Perform constraint suspension on the suspect.
    (c) If a contradiction is found, form conflict set, which reduces suspect set.
    (d) If no contradiction is found, add suspect to candidate set.
    (e) Remove suspect from unexamined suspect set.

Figure 3.1: The Original Diagnostic Algorithm

Value Propagation Costs

Value propagation costs fall into three categories:

- Binding the variables on the left-hand sides of behavior rules.
- Computing the conclusions of the rules, given the variable bindings.
- Recording the conclusion and maintaining dependency information.

A constraint network [Ste80] implements the propagation of values through the circuit. At each node of the circuit, the program keeps a list of behavior rules that can propagate a value from that node. When a new value is asserted at a node, each of the associated behavior rules checks if it is ready to fire (a multiplier rule, for example, is not ready until both of its inputs are asserted). This method is more efficient than pattern-directed rule invocation because the behavior rules do not need to search the entire database of assertions to find possible variable bindings.[3] All of the behavior rules require binding of approximately the same number of variables, so we approximate the variable binding costs for a rule firing as a constant, k_1. We count B, the number of behavior rule firings.

There is also a cost to computing the conclusions of the behavior rules, given variable bindings. For example, given the values at its inputs, a behavior rule for

[3]For historical reasons, our implementation is not quite this clean. Pattern-directed invocation is used for component rules, while the constraint network is used for wires and for detecting contradictions. As discussed in Section 3.8.1, the main inefficiencies in using pattern-directed rule invocation occur in propagating values through wires and detecting contradictions. Thus, our implementation eliminates the major sources of inefficiency in pattern-directed rule invocation.

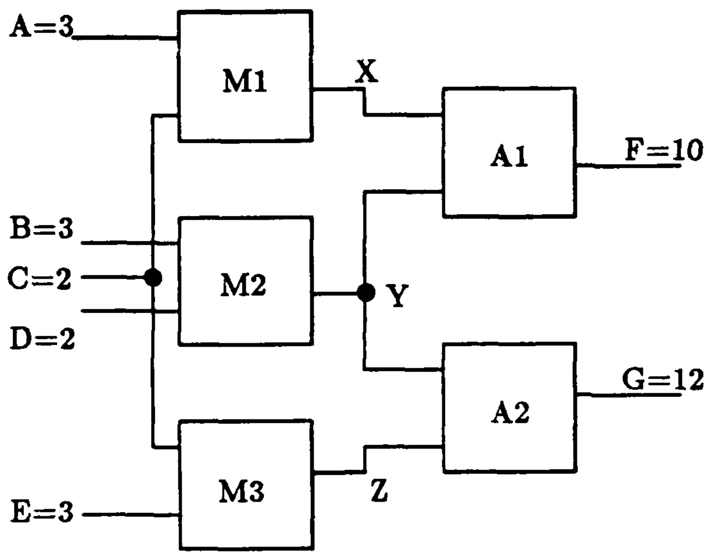


Figure 3.2: Polybox

M1 multiplies them. All of the component behavior rules for our circuits compute either single arithmetic or logic operations, which we estimate as a constant cost, k_2. Single arithmetic and logic operations take a negligible amount of time, relative to the other costs incurred, so k_2 will be negligible.

Finally, there are costs to asserting a new value. The program records dependencies so that it can find the components which supported derivations of contradictory values and also so that it can efficiently change its assumptions about which components are working. Each time a behavior rule is run, a justification is recorded stating that the antecedents of the rule support the conclusion. For example, if M1's forward behavior rule predicts the value 6 at X from 3 at A and 2 at C, the justification states that X is 6 as long as A is 3, C is 2, and M1 is assumed to be working. The conclusion is then placed in the database. In addition, each antecedent assertion must record that it is in a new justification. We make the simplifying assumption that all justifications involve the same number of assertions, so that the cost of recording a justification is a constant, k_3. One justification is installed for each behavior rule firing, so the cost of asserting a new value is $k_3 * B$.

$$C_{RULE-RUNNING} = (k_1 + k_2 + k_3) * B$$

Context Switching Costs

When the diagnostic engine performs constraint suspension on a component, it must suspend the assumption that the component is working. If the component has been used to predict values, those value assertions must also be removed. For example, if constraint suspension is performed on M1, the justification for asserting 6 at X is no longer valid. If that assertion is not supported by any other justification, the assertion must be removed. Removing that assertion may require the suspension of still further justifications and the removal of other assertions.

If constraint suspension is performed on M2 at some later time, the assumption that M1 is working will be reactivated. Our truth-maintenance system (a JTMS) caches (rather than erases) justifications that are no longer valid so that it can avoid re-running behavior rules. In this case, when the assumption that M2 is working is reactivated, the justification for 6 at X becomes valid again, and the assertion is brought back into the database, without re-running M1's behavior rule.

Each addition or removal of an assertion is caused by the activation or deactivation of a justification. The combination of the activation of one justification and the addition of one assertion, or the deactivation of one justification and the removal of one assertion, takes approximately constant time, k_4. Thus, the costs of switching contexts (sets of assumptions about which components are working) can be measured by counting TV (for truth-value changes), the number of assertions brought "in" to the database or removed from the database.

$$C_{CONTEXT-SWITCH} = k_4 * TV$$

Conflict Set Construction Costs

Once the diagnostic engine predicts contradictory values, it finds the components used in the derivation and notifies the system that there is a new conflict set. To find the components supporting the assertions of contradictory values, the program simply traces back through the justifications that the two contradictory assertions depend on. We approximate the cost of finding the components supporting a contradiction as a constant, k_5. Each conflict set must be recorded and intersected with the suspect set, incurring another constant cost, k_6. We count the number of conflict sets constructed, N.

$$C_{CONFLICT-SET} = (k_5 + k_6) * N$$

Cost Formula

The following formula summarizes the breakdown of costs incurred in first-principles diagnosis:

$$\begin{aligned} C &= C_{PROPAGATION} + \\ &\quad C_{CONTEXT-SWITCH} + \\ &\quad C_{CONFLICT-SET} \end{aligned}$$

1. Retrieve from the library for the device the generalized rules for noticing conflict sets. Check the applicability of each rule and intersect the conflict sets identified to form the initial suspect set.
2. If there are no conflict sets found using generalized rules, propagate values from inputs to outputs to find an initial contradiction, yielding an initial suspect set.
3. While there are still unexamined suspects:
   (a) Choose an unexamined suspect at random.
   (b) Perform constraint suspension on the suspect.
   (c) If a contradiction is found, form a conflict set, which reduces the suspect set.
   (d) If no contradiction is found, add the suspect to the candidate set.
   (e) Remove the suspect from the unexamined suspect set.
4. Use EBG to generalize each derivation of contradictory values found by propagating values and add the new rules to the library.

Figure 3.3: The Augmented Diagnostic Algorithm

$$\begin{aligned} = \; & (k_1 + k_2 + k_3) * B + \\ & k_4 * TV + \\ & (k_5 + k_6) * N \end{aligned}$$

3.4.2 With Generalized Rules

The augmented diagnostic algorithm uses all of the operations the original algorithm uses, but it also checks the applicability of generalized rules. Checking a generalized rule requires binding variables and checking preconditions, which we approximate as a constant cost operation. We count G', the number of generalized rules checked. When a generalized rule is applicable, a new conflict set is recorded, which incurs cost k_5. We count A', the number of conflict sets found using generalized rules.

Cost Formula

The following formula summarizes the breakdown of costs incurred in diagnosis aided by generalized rules:

$$\begin{aligned}
C' &= C'_{GENERALIZED-RULE-CHECKING} + \\
&\quad C'_{PROPAGATION} + \\
&\quad C'_{CONTEXT-SWITCH} + \\
&\quad C'_{CONFLICT-SET} \\
&= k_7 * G' + k_5 * (A' + N') + \\
&\quad (k_1 + k_2 + k_3) * B' + k_4 * TV' + \\
&\quad k_6 * N'
\end{aligned}$$

3.4.3 Cost Differential

The utility of generalizing derivations of conflict sets can be calculated as the difference in cost between diagnosis without using the generalized rules and diagnosis with the generalized rules. If the difference is positive, the generalized rules have improved the performance of the system.

$$\begin{aligned}
C - C' &= -C'_{GENERALIZED-RULE-CHECKING} + \\
&\quad (C_{PROPAGATION} - C'_{PROPAGATION}) + \\
&\quad (C_{CONTEXT-SWITCH} - C'_{CONTEXT-SWITCH}) + \\
&\quad (C_{CONFLICT-SET} - C'_{CONFLICT-SET}) \\
&= -k_7 * G' + \\
&\quad (k_1 + k_2 + k_3) * (B - B') + \\
&\quad k_4 * (TV - TV') + \\
&\quad k_5 * (N - N' - A') + k_6 * (N - N')
\end{aligned}$$

3.5 Breakdown of Results

It is clear from the last two columns of Figures 3.4 and 3.5 that the generalized rules improve performance because they reduce the number of behavior rule firings and context switches required during diagnosis. The reason is that, using the single-fault assumption, the program forms the initial suspect set by intersecting all of the conflict sets that the generalized rules identify. After that, the constraints imposed by some suspect will always be suspended, so it never performs the value propagations that require all of the initial suspects to be working. Without the generalized rules, on the other hand, some values are propagated assuming that all of those components are working, in order to identify the conflict sets. In Section 4.2.2 we will return to the key role played by the single-fault assumption, and argue that EBG cannot significantly significantly reduce the number of value propagations performed by a multiple-fault diagnostic engine such as GDE[dKW87].

	time in seconds	*G* Gen. rules checked	*A* Gen. rules Applic.	*N* New conflict sets	*B* Beh. rule firings	*TV* Truth-Value changes
No learning (training cases)	0.77	0	0	2.00	33.36	42.92
During learning (training cases)	0.56	2.96	1.97	.03	30.93	25.78
No learning (test cases)	0.76	0	0	2.00	32.49	42.44
After learning (test cases)	0.55	3.00	2.00	0.00	29.60	25.04

Figure 3.4: Results from Polybox Experiments. All numbers are averages over the 100 cases run.

	time in seconds	*G* Gen. rules checked	*A* Gen. rules Applic.	*N* New conflict sets	*B* Beh. rule firings	*TV* Truth-Value changes
No learning (training cases)	7.09	0	0	3.29	120.8	376.5
During learning (training cases)	6.23	134.0	2.04	1.47	115.4	289.1
No learning (test cases)	7.07	0	0	3.33	119.8	367.1
After learning (test cases)	5.66	221.0	3.15	0.56	112.5	248.2

Figure 3.5: Results from Adder Experiments. All numbers are averages over 150 cases.

3.6 Utility of Individual Generalized Rules

The equation makes it clear that the utility of an individual generalized rule in speeding up diagnosis depends on how frequently the rule applies, how expensive it is to check, and how much benefit is gained from it when it is applicable. The importance of a rule applying frequently is obvious, since all the terms except $-k_7 * G'$ drop out of the equation when the rule is not applicable: no benefit is gained from the rule when it is not applicable. One term in the cost differential formula is $-k_7 * G'$, which makes it clear that if checking a generalized rule is very expensive (i.e. k_7 is

$$\begin{aligned}
\text{benefit of applying once} &= \frac{\text{propagation time saved per case}}{\text{number of rules applicable per case}} \\
&= \frac{7.07-(5.66-.70)}{3.15} = .67\text{seconds} \\
\text{cost of checking once} &= \frac{\text{time to check all rules per case}}{\text{number of rules checked per case}} \\
&= \frac{.70}{221} = .0032\text{seconds} \\
\text{benefit/cost ratio} &= \frac{.67}{.0032} = 211
\end{aligned}$$

Figure 3.6: Derivation of benefit-cost ratio of a generalized rule based on results from the adder experiment.

high,) the utility of the rule may be low or even negative. Finally, the rule will have greater utility the more it reduces the number of behavior rules fired (B - B') and the context switching costs (TV - TV').

On average, the benefits gained from the applicability of a generalized rule to an adder case equaled the cost of checking a rule 211 times. That figure is derived in Figure 3.6. This means that generalized rules that, on the average, were applicable less than once every 211 examples slowed the system down. Moreover, if checking a generalized rule had been four times as expensive, or if there had been roughly four times as many, and all other factors remained constant, using the generalized rules would have slowed the system down overall. This analysis emphasizes that the benefits will not always outweigh the costs: the performance effect of the generalized rules is an experimental question.

3.7 Aggregate Analysis: Device Characteristics

The differential cost formula of Section 3.4.3 is also helpful in analyzing the effect of all of the generalized rules taken together. This leads to a characterization of the kinds of devices for which generalizing conflict set derivations will improve diagnostic performance. The devices that will gain the most are those in which only a few components ever fail, the behavior of components is inexpensive to compute, and the topology of the device is such that conflict sets tend to have few components in common.

The fewer different components of the device actually fail, the more effective this use of EBG will be. This is true because if only a few components of the device ever break, only a few patterns of value propagations will lead to pr^dictions of contradictory values. Hence, only a few generalized rules will be constructed, and the term $-k_7 * G$ will be small.

The simpler the component behaviors, the higher the utility of the generalized rules. Each generalized rule encapsulates a pattern of inferences, and checking the preconditions of a rule incurs all of the behavior costs of the rule firings encapsulated. For example, checking the precondition `(= ?F (+ (* ?A ?C) (* ?B ?D)))` in R1 involves the same two multiplications and one addition that would have been performed by the behavior rules for M1, M2 and A1. The arithmetic and logic operations performed by the components in the two circuits considered here are inexpensive, so the cost of checking the generalized rules is not prohibitive. Suppose the components computed square roots instead. Both k_7 and k_2 would be higher, so at first glance it is not clear what the effect on the utility of the generalized rules would be. The key point is that M1's behavior may be encapsulated in more than one generalized rule. Thus, checking the generalized rules would incur more square root computations than are saved by reducing the number of behavior rule firings (B - B'). Hence, the more expensive the component operations, the lower the utility of the generalized rules.

If the topology of the device is such that conflict sets tend to have few components in common, the utility of the generalized rules will be higher. This follows from the fact that the benefits gained from the applicability of a generalized rule depend on the number of suspects that it eliminates. The more suspects are eliminated, the greater the number of behavior rule firings (B - B') and truth-value changes (TV - TV') that are saved. If every conflict set includes several components that are not in any other conflict set, then each generalized rule, when it is applicable, will exonerate several suspects. On the other hand, if the device has n components and every conflict set has $n - 1$ components, each generalized rule that applies can only reduce the suspect set by one component. EBG will be most effective when the topology of the device is such that conflict sets have few components in common.

3.8 Aggregate Analysis: Alternative Diagnostic Engines

The differential cost formula of Section 3.4.3 also makes it clear that using EBG to generalize conflict set derivations may either improve or reduce diagnostic speed depending on the performance of the original diagnostic engine, including the relative costs of behavior rule firing, TMS operations, and checking generalized rules. For example, if checking a generalized rule is orders of magnitude more expensive than firing a behavior rule or performing a TMS operation, the generalized rules will cause a significant deterioration in performance. This section describes four changes to the implementation of the diagnostic program and discusses the effect each would have on the utility of this use of EBG. The overall theme is that reducing the cost of rule firing and context switching reduces the benefits, while reducing the cost of checking generalized rules increases the ability of EBG to improve diagnostic performance.

3.8.1 Pattern-Directed Rule Invocation

If our diagnostic engine used pattern-directed rule invocation, EBG would improve performance even more than in the experiments. The reason is that pattern-directed rule invocation is less efficient than a constraint network for propagating values in a circuit, so using pattern-directed rule invocation would increase the size of k_1, the cost of matching the preconditions of a behavior rule. An earlier version of our diagnostic program in fact did use pattern-directed rule invocation to trigger behavior rules. Using that version, the total time saved using the generalized rules was greater than the time saved using the generalized rules with the constraint network implementation. The experiments are reported in Appendix B.

To understand why pattern-directed rule invocation is less efficient than a constraint network for value-propagation in circuits, consider the wire rule below. The assertion of a new value anywhere in the circuit causes the pattern-matcher to attempt to match the new value with *every* wiring assertion in the database. The constraint network, on the other hand, can find by one-step lookup the wires connected to the location at which the new value is asserted.

```
(defrule WIRE-EQUALITY-FORWARD
  IF (and [wire ?terminal1 ?object1 ?terminal2 ?object2]
          [value-of ?terminal1 ?object1 ?value])
 THEN
  (assert [value-of ?terminal2 ?object2 ?value]))
```

Similarly, a pattern-directed rule that detects when two contradictory values are asserted at the same location will check each new value asserted against every value asserted anywhere in the circuit. The constraint network, on the other hand checks only those values asserted at the appropriate location.

3.8.2 Using an ATMS

Using an ATMS [dK86] rather than our JTMS would eliminate the context switching time during diagnosis, at the expense of increasing the time to run rules and record conflict sets. As a result, it is not clear whether generalizing conflict set derivations would have more or less utility in speeding up an ATMS implementation of the diagnostic engine. Instead of "inning" and "outing" assertions from a database when different contexts are considered, the ATMS keeps track of the minimal contexts in which any assertion can be supported. If a predicted value can be supported in two ways, two minimal contexts are recorded for it. When a conflict set is found, rather than retracting the assumption that some component is working, the ATMS leaves the database intact, but refuses to run any more behavior rules in contexts that include all of the components of the conflict set.

We implemented a version of the ATMS candidate generator in GDE [dKW87], modified to take advantage of the single fault assumption. The modified version refuses to run behavior rules in contexts containing all of the components in the suspect set (the intersection of the conflict sets found so far). Using the ATMS eliminates context switching costs, but the cost of running a rule is higher than with a JTMS because the context label of the conclusion of the rule must be updated. Recording the labels also incurs a space cost: in diagnosing the lookahead adder circuit, more than 3600 different labels had to be stored. In addition, recording a conflict set is more expensive because all of the context labels containing the conflict set must be updated. Overall, EBG improved performance in diagnosis of the polybox circuit, but we were not able to evaluate how effective EBG is in speeding up diagnosis of the adder circuit because the diagnostic engine was too slow to permit experiments in the time available.

3.8.3 Keeping Dependencies Only

An alternative diagnostic engine might rerun behavior rules rather than use a TMS to cache deductions. This might reduce the cost of context switching, and hence reduce the benefits of using EBG. Our implementation uses a JTMS both to keep track of dependencies during value propagation and to cache behavior rule firings so that they do not need to be re-run when the diagnostic engine changes its assumptions about which components are working. However, running behavior rules is not very expensive for circuits such as polybox and the lookahead adder, because the constraint network makes triggering a behavior rule cheap, and evaluating the body of a behavior rule involves only a single arithmetic or logic operation. An alternative diagnostic engine might simply clear all of the values asserted in the circuit and propagate anew when it performs constraint suspension on a new suspect. That might be faster than using a JTMS to remove assertions from the database and add others in. If context switching in the alternative diagnostic engine were faster than in our diagnostic engine, the benefits from using EBG would be reduced.

3.8.4 Reducing Costs of Checking Generalized Rules

The utility of EBG would be higher if the generalized rules could be checked less expensively. The time necessary to check generalized rules can be reduced by sharing some computation, as described below.

Sharing variable bindings can reduce the cost of checking generalized rules. All of the generalized rules use only the observed values of the inputs and outputs of the circuit. Some of the inputs and outputs are used in checking more than one generalized rule. If the variables representing the input and output observations were bound once and then used in checking all of the generalized rules, rather than binding variables separately when checking each generalized rule, the time necessary to check all of the generalized rules would decrease.

Some computations are repeated in more than one generalized rule, and those computations could also be shared. For example, the expression `(* ?A ?C)` appears in both R1 and R2. Since multiplication is very inexpensive, it probably would not pay to store the product and use it to check both R1 and R2. If more expensive operators appeared in the rules' preconditions, however, which would occur if the circuit components' behavior were more expensive to compute, sharing the computation of common subexpressions might reduce the cost of checking the generalized rules. This is analogous to the Rete Net idea of sharing the evaluation of predicates that appear in multiple rules. Here, however, what would be shared is the evaluation of subexpressions that are part of different predicates.

3.9 Trading Off Precision for Speed

One direction for future research is to explore the effects of trading off precision for speed. The augmented diagnostic system is guaranteed to produce the same candidates that the original diagnostic engine would produce. The generalized rules never construct incorrect conflict sets, so no components are mistakenly exonerated. The system performs constraint suspension on any components that are not exonerated by generalized rules, so every component that can be exonerated is. As the program accumulates experience, more suspects are exonerated by generalized rules and hence fewer are exonerated by constraint suspension. At the risk of missing the exoneration of a few components, an optimistic augmented algorithm could skip the constraint suspension step once it had accumulated a large library and deem any component in the initial suspect set a valid candidate. The proposed candidate generator would produce correct diagnoses, but perhaps less precise ones than are possible.

Table 3.1 summarizes the speed gained and the number of extra candidates generated using generalizations from more and more training examples. The more training examples were used, the more precise the final candidate sets were. More results can be found in Appendix B. Of course, the effects of trading off precision for speed depend on the larger diagnostic context. Can the extra hypotheses be eliminated easily at the next stage, or do they prove very costly? That question is beyond the scope of this research.

3.10 Conclusion

Our experimental results demonstrate that EBG can improve diagnostic performance on both the polybox circuit and the adder circuit. The differential cost formula of Section 3.4.3 makes it clear that the change in speed resulting from this use of EBG depends on characteristics of the circuit and on the relative costs of rule running, TMS operations, and checking generalized rules. If the device has only a few failure modes, has component behaviors that are inexpensive to compute, and

Training examples used	No. of gen's. constructed	Time (w/o gen's.)	Time (w/ gen's.)	No. of candidates (w/o gen's.)	No. of candidates (w/ gen's)
100	181	7.15	.65	4.69	8.83
150	221	7.06	.70	5.13	7.35
200	245	6.99	.75	5.16	6.78

Table 3.1: Speed vs. precision tradeoff in diagnosis

a topology such that the conflict sets have only a small overlap, the use of EBG may speed up the system significantly. Reducing the cost of running behavior rules and switching contexts for the original diagnostic engine will reduce the utility of using EBG to generalize conflict set derivations, while reducing the cost of checking generalized rules will increase the utility.

Chapter 4

The Sources of Power in EBG

The learning program that was analyzed in the previous chapters used EBG to construct generalized rules that could recognize when a previous derivation of a conflict set was applicable to a new set of observations. In later chapters, EBG will be used to generalize based on other kinds of similarities. In this chapter we step back from the particular uses to examine EBG as a tool.

We analyze in turn the two most common uses of EBG, generalizing successful problem solving episodes and generalizing explanations of failures. For each technique, the sources of power of EBG are analyzed. One conclusion of the analysis is that no "operationality criterion" can guarantee that all of the generalized rules that are constructed will have a beneficial effect on problem solving speed. The analysis of each technique culminates in a qualitative characterization of the problems for which EBG is likely to improve performance.

The analysis in this chapter is motivated both by the analysis of the experimental results in the previous chapter and by previous research that demonstrates that EBG will sometimes but not always improve performance [Min85, Min88, TN88]. Previous research has tried to understand the effect of EBG on performance by analyzing the utility of individual generalized rules [MCE$^+$87, Min88, TN88]. By contrast, we believe there are characteristics of problem formulations, problem solvers, and distributions of examples that will affect the utility of *all* of the generalized rules. We try to expose them by analyzing the effects of EBG in terms of changes to a problem solver's search strategy.

Throughout the chapter, problem solving is formulated as search. The problem solver starts with an *initial state* and a set of *operators* that create new states (or search nodes) from ones that have been constructed already. The operators have preconditions that indicate which states they can be applied to. Normally, the problem solver's task is to find a single state that satisfies the given goal criteria; alternatively, the task may be to find *all* of the states in a finite search space that satisfy the goal.

As in the previous chapter, we ignore the cost of creating generalized rules and focus only on the cost of checking them, because we assume that the problem solver can amortize the cost of generating a rule over a large number of cases.

4.1 Generalizing Successful Search Paths

This section analyzes the most popular use of explanation-based generalization, generalizing successful search paths. The problem solver remembers and encapsulates a sequence of operator applications that led to a goal state in solving one problem. In solving each future problem, it checks whether that operator sequence is applicable (the preconditions of all the operators are satisfied) and leads to a goal state. If no remembered operator sequence is useful, it falls back on its original techniques for searching the space. STRIPS' construction of macro-operators [FHN72] and SOAR's chunking [LNR87] are two other mechanisms for encapsulating successful sequences. While they use different mechanisms, the resulting generalizations are the same as those that would be constructed using EBG [RL86].

There are two sources of power in using EBG on successful search paths. The first source of power results from remembering useful patterns of inferences, so that search in future problems is biased towards paths that have been useful previously. This source of power rests on two assumptions: first, some patterns of operator applications are useful more frequently than others; second, the distribution of future cases that the problem solver will be presented with is reflected by the distribution of cases it has seen so far. The second source of power results from encapsulating the search paths, so that the problem solver can jump to the conclusion of a remembered pattern of inferences without constructing any of the intermediate search nodes. Briefly put, using EBG on successful search paths allows the problem solver to find good search paths quickly (search bias) and to travel down those paths quickly (encapsulation).

4.1.1 Biasing Search Toward Previously Successful Paths

The first potential for speedup from remembering successful search paths in the form of generalized rules is that the rules can be used to bias search towards paths that were successful before, thus reducing search. That bias will be more effective in improving performance the more paths the original problem solver explores that *never* lead to a solution. In fact, performance may actually deteriorate if too many search paths lead to a solution, even if most of them lead to a solution only rarely. At the end of the section we propose propose additional mechanisms to use with EBG so that the search bias can improve performance when there are many paths that *rarely* lead to a solution, but few that *never* lead to a solution.

Each time the problem solver checks the applicability of a generalized rule, it is as if the problem solver were exploring the search path which the rule generalizes. Thus, EBG alone biases search towards paths that led to solutions before and away from those paths that have never led to a solution.

To illustrate the weakness of using EBG alone to bias the search, suppose that, due to some manufacturing defect, component M1 causes 99% of the failures in the polybox circuit and component M3 accounts for 1% of them. The troubleshooter is presented with 100 cases, 99 that resulted from M1 failing and one that resulted from M3 failing. It generates rule R1 for recognizing that the derivation of the conflict set `(M1 M2 A1)` is applicable and uses R1 98 times. It generates rule R3 for recognizing that the derivation of the conflict set `(M2 M3 A2)` is applicable but R3 never applies again. Yet after solving those 100 "training instances," R1 and R3 have equal status. In diagnosing future cases, R1 will be useful very frequently, while checking R3 will be a waste of time in 99 cases out of 100.

More generally, EBG will bias search most effectively when many of the possible search paths *never* lead to a solution. Remembering and using every successful pattern of inferences amounts to representing recurrence by a single bit: "Have I seen a problem like this *at least once* before." Search in solving future problems is biased toward paths that have been useful before and away from paths that have *never* been useful before. In the worst case, when every possible search path has led to a solution at least once before, all search paths will have equal status, and the search degenerates into a blind generate and test. If the original problem solver was able to do better than a blind search, use of EBG in this worst-case scenario could slow down the problem solver considerably. In short, EBG will bias search most effectively when there is a bimodality in the frequency with which search paths leads to goal states: every search path should lead to a goal state either frequently or not at all.

One way that search could be biased more effectively is for the program to keep some information about how frequently generalized rules are applicable. Several strategies are possible, of which we outline two. First, the program could keep track of the exact frequency of applicability of each generalized rule. Using the frequency statistics it could order the checking of rules so that rules that were useful more frequently would be checked first. It could also "forget" rules that were useful too infrequently. Second, instead of keeping explicit statistics, it could keep a fixed number of generalized rules in a Least Recently Used queue, bringing a rule to the front of the queue when it is used and throwing out the least recently used rule when the queue is full. Section 4.4.3 describes more elaborate statistical mechanisms, actually implemented in [Min88], that take into account the cost of checking a rule and the benefits derived from its being useful, as well as how frequently it is useful.

One possible improvement to the use of EBG that does not involve additional statistical mechanism is to make judicious choices as to which successful solution paths should be packaged up into generalized rules. Bottom-up chunking [Ros83, RN86] is one method for making those choices. Bottom-up chunking assumes that there is a hierarchy of problem spaces. That is, the problem solver can set up new search spaces to solve sub-problems while it is working on a larger problem. The bottom-up chunking method then chooses to encapsulate a successful search path only if the problem solver did not need to create and solve any sub-problems while taking that search path. Thus, the first time the problem solver solves a difficult problem, it will create generalized rules (chunks) from the solution of the lowest-level subproblems. If it solves the same problem again, those generalized rules will allow the problem solver to avoid creating the lowest-level subproblems, and it will create generalized rules for the next level of problem solutions. In this way, the choice of which generalized rules to create from the solution of a particular problem depends not only on the current problem but on all of the previous problems presented to the problem solver. Chunking in SOAR, however, no longer uses the bottom-up approach.

Summary

The straightforward use of EBG to remember every successful search path will bias search to the extent that many paths never lead to solutions. Because EBG considers each case in isolation, it fails to distinguish patterns that were frequently useful from those that were rarely (though sometimes) useful. One implication of this is that there is no way of filtering out "non-operational" generalized rules as they are created: the utility of a generalized rule depends on characteristics of the whole distribution of examples, not just the characteristics of any one example. If there are many paths that lead to solutions rarely (but sometimes), EBG may need to be embedded in a learning system that pays attention to the whole distribution of cases.

4.1.2 Encapsulating Patterns of Operator Applications

A second potential source of speedup from generalizing successful search paths is that EBG *encapsulates* a pattern of operator applications. That is, it remembers only the weakest preconditions for a search path and its conclusion rather than the whole path. The encapsulation makes it possible to check whether a whole sequence of operators can be applied and will lead to a goal state, and then jump to that goal state, all without actually applying any of the operators.

Two factors can make checking the weakest preconditions and then jumping to the final state more efficient than simply re-running all the operators. One factor can make it less efficient. Some operators compute their conclusions based on the variable bindings for their left-hand sides (e.g., M1's behavior rule performs one multiplication to compute its conclusion). We call that computation the *behavior* costs of the operator.[1] As we will see, all of the behavior costs of an operator sequence

[1]In applications involving only logical inference from boolean assertions (e.g. Winston's cup example [WBKL83]), the behavior cost of every operator is zero, because operators' right-hand sides are not functions of the variables on their left-hand sides.

are paid when checking the weakest preconditions. On the other hand, the *overhead* cost of matching the operators' left-hand sides and of recording their conclusions are eliminated when checking the weakest preconditions. The first positive factor, then, is the savings in overhead: the cost of matching the operators' preconditions and constructing intermediate search states. A second positive factor is that the weakest preconditions may be simplified, saving some of the behavior costs that are encapsulated in those preconditions. On the negative side, checking several generalized rules may repeatedly incur the same behavior cost that would have been shared among several search paths, had they not been encapsulated. Overall, the smaller the cost of evaluating the bodies of operators, relative to the cost of matching operator preconditions and storing results, the greater the benefits from encapsulation will be.

Saving Overhead

Generalized rules encapsulate the firing of several behavior rules, thereby saving the overhead of running them. Given specific values for A, B, C, and D, consider the difference between evaluating the expression `(+ (* A C) (* B D))`, from rule R1, and propagating the inputs through M1, M2 and A1 of the polybox circuit. In either case, two multiplications and one addition are performed. However, there is some overhead cost associated with propagating the values through the components. One source of overhead is rule triggering: each operator's preconditions are checked and its variables are bound. A second source of overhead is recording the conclusion of the operator (e.g., that the output of M2 is 6). Depending on how high these overhead costs are, and how long the encapsulated operator sequences are, saving the overhead costs of the encapsulated behavior rule firings may be significant.

Expression Simplification

A second potential source of efficiency is simplifying a generalized rule's preconditions, thus saving some of the behavior costs as well as the overhead costs of re-running the operator sequence. Consider the left-hand side of R5, from the adder circuit. S3 was propagated back through X3, assuming X3's other input was 0, and then through N24, yielding the expression `(INVERT (XOR ?S3 0))` which appears in the precondition `(NOT (= (INVERT (XOR ?S3 0)) 0))`; evaluating the equivalent precondition `(NOT (= ?S3 1))` would save the behavior costs of firing X3's and N24's behavior rules. The implemented program does not perform any such behavior simplifications, partly because the behavior costs for adders, multipliers, and gates are very small. Although unguided expression simplification is a hard problem, it would be worth using a MACSYMA-like system to simplify the generalized rules' preconditions as much as possible, whenever the behavior costs of operators are high.

Additional Behavior Costs

One negative effect of encapsulating patterns of behavior rule firings is that the same behavior rule firing may be encapsulated in several generalized rules. For example, in deriving conflict sets by propagating values through the polybox circuit, the prediction of the value 6 at the output of M1 was used in deriving both conflict sets, `(M1 M2 A1)` and `(M1 M3 A1 A2)`. Thus, the expression `(* A C)` appears in both R1 and R2 and is evaluated twice in diagnosing a new case, yet the same expression is evaluated only once when deriving the two conflict sets by propagating values, because the product is stored as the output of M1. More generally, checking the preconditions of the generalized rules will repeatedly incur the behavior costs of any shared rule firings.

Sufficient repetition of behavior costs in the preconditions of generalized rules makes it worthwhile to share those costs. Analogous to the idea of rete networks, subexpressions common to more than one generalized rule could be computed just once, the smallest ones first. That would eliminate the repetitive computation, but would require storing the intermediate results of subexpression evaluation. Keep in mind that one overhead saving from encapsulating proof trees is avoiding the storage of intermediate results (the other is avoiding the binding of variables for the operators). However, if the behavior costs of the operators are sufficiently high, eliminating repetitive computation of those behavior costs will be worth the re-introduction of the overhead of storing intermediate results.

4.1.3 Finding All Solution States

All of the above discussion about biasing search and encapsulation becomes moot if the problem solver has to find *all* of the goal states in a finite search space, rather than a single goal state. In searching for all of the goal states, there is no reason to check generalized rules unless the program is assured that its set of generalized rules covers all of the possible solution paths. To see this, note that after finding some solutions using generalized rules, the program still exhaustively explores the search space to find other potential solution states. Figure 4.1 graphically illustrates this problem. Even after S1 and S2 are found using generalized rules, nodes A, B and C must be constructed and visited in order to find S3 and S4 and check whether they are solutions. Thus, the problem solver loses both the search bias effect (it explores the rest of the search space) and the encapsulation effect (it constructs A and B).

4.1.4 Summary of Problem Characteristics

There are two sources of power in Explanation-Based Generalization of successful problem solving episodes. First, the generalized rules can act as remembered patterns of operator applications to bias the problem solver toward patterns that have

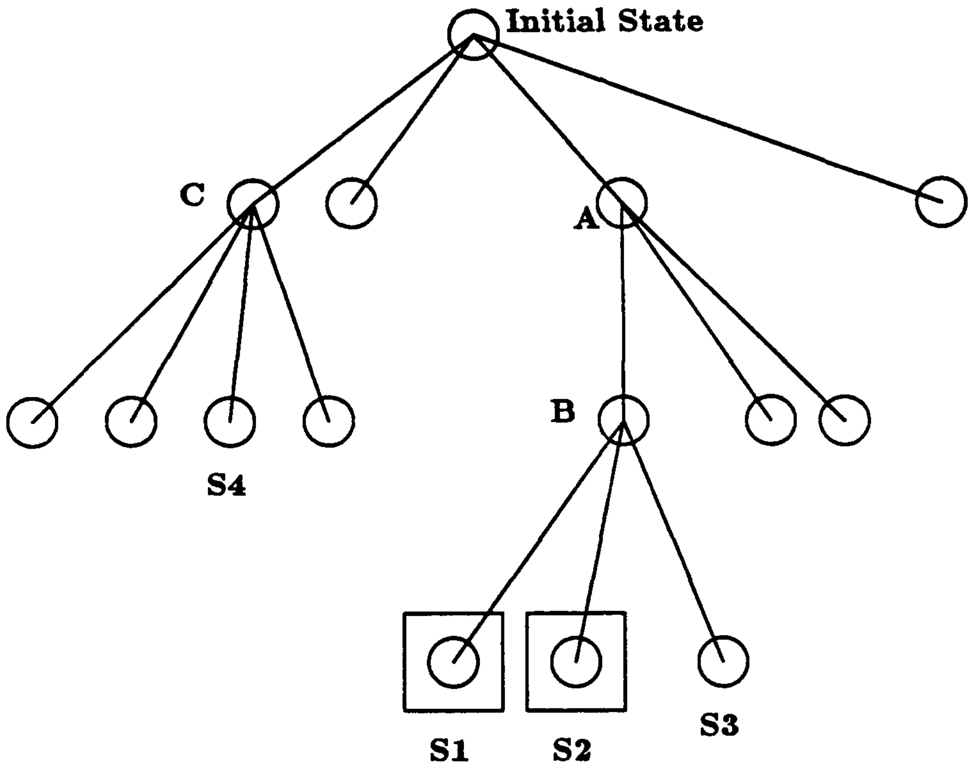


Figure 4.1: The problem solver must find all solutions. S1 and S2 are found using generalized rules; nodes A, B, and C must be constructed and visited in order to find S3 and S4 and check whether they are solutions.

been useful in solving previous problems. Second, the generalized rules encapsulate the patterns of operator applications, so that the overhead cost of checking operator preconditions and constructing intermediate search nodes can be eliminated, at the expense of evaluating the bodies of some operators more than once. This section summarizes the characteristics of problems for which EBG alone will produce a significant improvement in problem solving speed.

Bimodal Distribution The more potentially useful patterns of operator applications that are *never* actually useful in solving a problem presented to the problem solver, the more effective EBG will be in improving performance.

Inexpensive Rule Bodies The smaller the cost of evaluating the bodies of operators, relative to the costs of matching operator preconditions and storing results, the more effective EBG will be in improving performance.

Search For One Solution The problem solver's task must be to search for a single goal state, not all goal states, unless it can be sure that its generalized rules encapsulate all of the possible search paths.

4.2 Generalizing Explanations of Failures

A second popular use of explanation-based generalization in problem solving is to encapsulate the explanation of why a search node is *inconsistent* with a goal state. A generalized rule may be used either to speed up the process of checking whether a node satisfies the goal conditions (by ruling it out quickly), or to prune other nodes from the search space, or both. This section considers each of the two uses in turn.

4.2.1 Finding the Failure Faster

One way that a generalized explanation of a search node inconsistency can be used is to reduce the effort the problem solver expends in testing other search nodes for consistency. The problem solver can use a generalized rule to identify a search node as inconsistent faster than it would have been able to find the inconsistency without the generalized rule.

This is effectively using EBG to encapsulate *successful* patterns of operator applications (discussed in Section 4.1), provided we reformulate proving the *inconsistency* of a node with the goal state as a search problem in a separate space. The operators in this search space are inference rules, the initial state is a set of assertions about the node from the original space, and the goal is to derive a contradiction. The generalized rule that is created encapsulates a successful deduction of a contradiction.

We can draw on the analysis of Section 4.1 to characterize when speed will increase by using EBG to generalize the deduction of the inconsistency of a search node. First, in order for EBG to bias search, there must be a bimodal distribution in the frequency of applicability of derivations of inconsistency: derivations must be applicable frequently or not at all. Second, to benefit from encapsulation, the bodies of the inference rules used to derive the inconsistency of a search node must be inexpensive to evaluate. To the extent that these hold, EBG can help the problem solver to prove failures faster.

4.2.2 Reducing Search

A second way to use generalized rules that identify inconsistent nodes is to reduce search in the original space. First, by assuming that inconsistent search nodes never lead to goal states, the problem solver can cut off the entire sub-space reachable from the node identified as inconsistent. That excision will not improve performance, however, if the original problem solver was able to cut off the same sub-space. Second, by using explicit simplifying assumptions, the problem solver may be able to ignore an even larger sub-space.

The inconsistency of a search node can be used to cut off search only if inconsistency is *monotonic* (i.e. no goal state can ever be reached from a state that is inconsistent with the goal [MB87]). Otherwise, the inconsistency of a search node justifies avoiding only that node during the search, not any of its successors. Planning problems normally do not satisfy the monotonicity of inconsistency criterion, while constructive problems such as design and local constraint satisfaction problems do satisfy it. In planning problems, some operators tend to recover from the effects of others, so even if a state is far from satisfying the goal criteria, the problem solver can not assume that no goal state is reachable from there. In some constructive problems, on the other hand, such as design, additional operator applications simply add to the design, rather than change it, so a partial design that is inconsistent cannot lead to a complete design. Local constraint satisfaction problems [Mac87], where the problem solver's task is to assign labels to a number of objects without violating a set of constraints among the labels, also satisfy the criterion: if a partial labeling is inconsistent, every complete labeling resulting from it will also be inconsistent.

Cutting off a Search Sub-tree

Assuming the monotonicity of inconsistency, a generalized rule that identifies a search node as inconsistent can be used to excise the entire sub-tree reachable from that node. This can offer a significant savings if the original problem solver would have explored that sub-tree.

However, in order to make a fair comparison, the original problem solver should also be allowed to make use of the monotonicity of inconsistency to cut off search at nodes that it proves inconsistent. As we explore below, in solving constraint satisfaction problems and performing circuit diagnosis, a good problem solver can and should check the consistency of each node as it is visited. In that case a generalized rule can *not* significantly reduce the search. It may, however, still be useful in proving quickly that individual nodes are inconsistent, as in Section 4.2.1.

The original problem solver may not be able to exploit the monotonicity of inconsistency if checking for inconsistencies in partial solutions is much more expensive than checking for inconsistencies in complete solutions. Consider analog circuit design to meet certain global speed and power usage requirements [Wil88]. It is very expensive to check a partial design for consistency with the requirements, but completed designs can be simulated. Hence, using EBG to explain failures of circuit designs may be very useful in speeding up later design performance.

In solving local constraint satisfaction problems, however, the problem solver should check the consistency of partial labelings rather than waiting until it has constructed complete labelings. Consider, for example, the Failsafe program [MB87], which learns from explanations of its failures in solving simplified floor planning problems. Its task is to place a given set of rectangles (rooms) of given sizes onto a larger rectangle (the floorspace), such that the placement satisfies certain constraints, such as rooms not overlapping. This can be viewed as a constraint satisfaction problem, where each room must be labeled with its position on the floor. The program uses a generate and test exploration of the problem space, placing all of the rooms and then checking if the placement satisfies all of the constraints. Failsafe's performance improves because the generalized rules it constructs from the explanations of the inconsistency of room placements prune a large portion of the search space that the generate and test problem solver would have explored. If, however, the original problem solver had checked the consistency of partial labelings, the generalized rules would prune only portions of the search space which the original problem solver *would have avoided.* Hence, using EBG to identify inconsistent partial solutions to constraint satisfaction problems only eliminates search subtrees that the original problem solver should not have explored in any case.

A second example of the original problem solver being able to cut off search at inconsistent nodes is provided by our use of EBG to generalize the derivation of contradictory values during model-based diagnosis. We formulate the diagnostic engine's task as search through the space of subsets of components (contexts), with the goal of finding the largest subsets that are consistent with the observations. The search operators are *not* the component behavior rules, but rather are operators that add one additional component to a context. The component behavior rules are used to derive contradictory values in a context, thus identifying the context as inconsistent. Generalized rules also identify inconsistent contexts. The monotonicity of inconsistency criterion is satisfied: if M1, M2, and A1 can predict contradictory values at F, any larger set of components can predict the same values at F. However, a diagnostic engine should check for the inconsistency of contexts as it visits them, and the one we used does so. If that is the case, the diagnostic engine cuts off search below inconsistent nodes, with or without the generalized rules. Thus, the diagnostic engine of Chapter 2 is another problem solver for which the generalized rules don't do any better than the original problem solver at cutting off search below inconsistent nodes. As will be described shortly, however, the generalized rules allow it to cut off search above inconsistent nodes, which the original problem solver could not do.

Cutting off Search Above the Failure Node

By using simplifying assumptions, it may be possible to combine the results of two or more generalized rules to cut off search above the nodes that generalized rules identify as inconsistent. Section 3.5 discussed using the single-fault assumption to intersect the conflict sets identified using generalized rules. The single-fault assumption in diagnosis is one example of a simplifying assumption that allows the problem solver to combine the results of several generalized rules.

The single-fault assumption allows the diagnostic program to cut off search even before it reaches the contexts the generalized rules identify as inconsistent. For example, in Figure 4.2, if two generalized rules identify the conflict sets (B1 B2 B3 B4) and (B1 B2 B57) for a hypothetical circuit with 57 components, the augmented di-

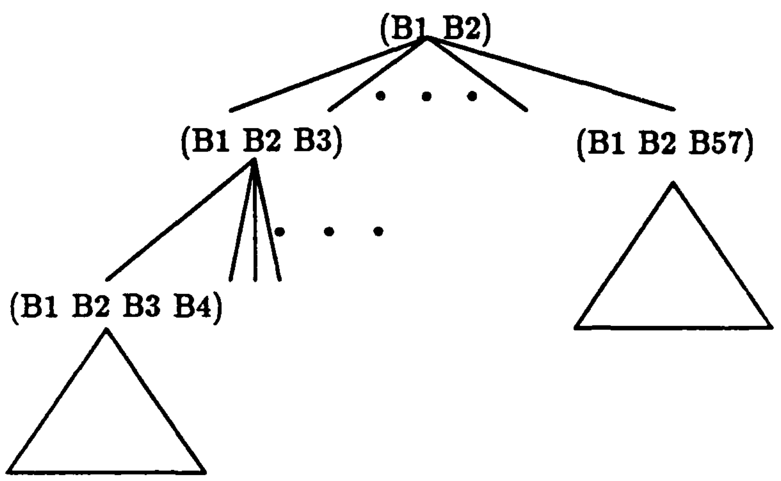


Figure 4.2: With or without EBG, the problem solver can cut off the subspaces rooted at the inconsistent contexts (B1 B2 B3 B4) and (B1 B2 A57). Using EBG and the single-fault assumption to intersect inconsistent contexts, the problem solver can avoid the entire subspace rooted at (B1 B2).

agnostic engine intersects the two conflict sets, to reduce the suspect set to (B1 B2). When propagating values, it never explores any contexts containing both B1 and B2 (i.e. it never propagates values using both B1 and B2.)

Using the single-fault assumption to intersect inconsistent contexts is a special case of the use of explicit simplifying assumptions to combine the results of generalizations. Explicit ONE-OFs and the following hyperresolution inference rule, taken from [dKW86], provide a more general framework for combining the results of several generalized rules that identify inconsistent search nodes.

$$\frac{\begin{array}{l}\text{ONEOF}(A_1, A_2, \ldots) \\ \text{CONFLICT-SET}\alpha_i \text{ where } A_i \in \alpha_i \text{ and } A_{j \neq i} \notin \alpha_i \text{ for all } i\end{array}}{\text{CONFLICT-SET } \bigcup_i [\alpha_i - \{A_i\}]}$$

The single-fault assumption is equivalent to stating that, given any pair of components, one of them must be working. That is, there is a ONEOF for every pair of components. Instantiating the hyperresolution rule above with the assumption (ONEOF B4 B57) and the conflict sets (B1 B2 B3 B4) and (B1 B2 B57), identifies the conflict set (B1 B2 B3). This, together with conflict set (B1 B2 B57) and the assumption (ONEOF B3 B57), can be used to identify the conflict set (B1 B2). The effect is the same as intersecting the two original conflict sets, but the process is less efficient, because the program may try many choices of ONEOFs before finding the right ones to use with the hyperresolution rule.

While hyperresolution is not as efficient as set intersection for exploiting the single fault assumption, hyperresolution is a more general framework. Hyperresolution can exploit weaker simplifying assumptions than the single fault assumption. The simplifying assumptions, together with hyperresolution, could be used to cut off search above the search nodes that the generalized rules identify as inconsistent. Such a use of generalized rules would reduce the search space that the problem solver explores, potentially improving its performance.

4.2.3 Summary of Problem Characteristics

There are two potential sources of power in using EBG to encapsulate explanations of the inconsistency of search nodes. First, finding the inconsistency of a search node may be very expensive, in which case encapsulating a successful derivation of an inconsistency may reduce that cost. In that case, generalizing explanations of failures is the same as generalizing *successful* patterns of inferences in the space of derivations of inconsistencies. Second, knowing the inconsistency of one search node may enable the problem solver to ignore a large portion of the search space. Several characteristics of problems make them appropriate for generalizing the explanations of failures to reduce the search space:

Monotonicity of Inconsistency If the the inconsistency of a search node with the goal conditions implies that no goal state can be reached from it, the problem solver can cut out the search sub-spaces rooted at that node. Consistent labeling problems and constructive problems such as design may have this characteristic, but planning problems generally do not.

Relative Cost of Checking Inconsistency Checking for the inconsistency of a partial solution should be much more expensive than checking the consistency of a complete solution. Otherwise, the original problem solver will check the consistency of each node as it constructs it and the generalized rules will prune only parts of the search space that the problem solver would not have explored in any case.

Simplifying Assumptions If reasonable simplifying assumptions (such as the single-fault assumption for circuit diagnosis) can be used to combine the knowledge that several search nodes are inconsistent, EBG may speed up problem solving by cutting off search at nodes above the nodes identified as inconsistent by the generalized rules.

4.3 A Note on Parallelism

Some researchers have suggested that parallel processing will reduce the marginal cost of checking additional generalized rules to zero. If the cost of checking a generalized rule were zero, then the analysis in this chapter would be moot: it does not matter how much benefit a generalized rule provides because there is no cost to using it. The argument, however, is not correct, because it considers only clock time, and not processor time, as a resource.

While the analysis in this chapter is phrased in terms of time costs, it could easily be generalized to refer to arbitrary resource costs. Current research in parallel processing considers processor time an important resource. As evidence, consider that some complexity analysis of parallel algorithms is parameterized by both n, the size of the problem and by p, the number of processors to be used [Sch80]. Other complexity analysis of parallel algorithms explicitly gives both the step-complexity (clock-time) and the element-complexity (the total amount of processor time used) [Ble88].

If processor time is considered a resource, then the cost of checking all of the generalized rules will always grow with the number of generalized rules, and the analysis in this chapter is relevant. Another way to look at this is to note that it is misleading to use a parallel processor to check the generalized rules without considering if the original problem solver could have made better use of those processors. Even with a parallel processor, we must be careful about whether the generalized rules are improving or reducing performing.

4.4 Related Work

4.4.1 The Causes of Expensive Generalized Rules

Tambe and Newell [TN88] have analyzed characteristics of problem spaces that lead to the creation of individual generalized rules (chunks) that are expensive to check. Their research is useful since one factor that affects the utility of using generalized rules is the cost of checking them. Expensive generalized rules, however, may be the ones that are most frequently applicable, or the ones that provide the most benefit when they are applicable. In this document we have tried to identify characteristics of the problem space which will lead to the creation of rules that have positive utility overall.

4.4.2 Operationality Criterion

Most previous work on evaluating the utility of EBG has focused on finding an "operationality criterion" for selecting only those generalized rules that will speed up performance. The analysis in this chapter is novel in that it recognizes that no such criterion is possible, and instead uses the factors affecting operationality (recurrence, manifestness, and exploitability) to characterize the *problems* for which EBG will improve performance.

The search for a good operationality criterion led to identifying first manifestness, then recurrence and exploitability, as the key factors affecting the utility of a generalized rule. Originally, the only issue considered was how manifest the generalizations would be. Early work sought simple restrictions on which predicates could appear in the preconditions of generalized rules [MKKC86]. Unfortunately, restrictions on predicates turned out to be insufficient to capture the requirement that generalized rules be checked efficiently. For example, the predicate PROVABLE can be evaluated efficiently on the theorem "2 + 2 = 4" but not so easily on Fermat's last theorem [DM86]. Later work has fallen back on measuring the cost of checking generalized rules after they are constructed, as we did in Chapter 3.

More recently, work on PRODIGY identified the notions of recurrence and exploitability [MCE$^+$87] and research on MetaLex [Kel87a, Kel87b] focused on exploitability. The recurrence of a rule depends not only on how many cases it applies to (the generality criterion of [Seg87]), but also the frequency with which those cases occur in the distribution of cases presented to the problem solver. Keller pointed out [Kel87b] that the degree to which a problem solver can exploit a generalized rule may change over time as the problem solver acquires more rules.

The importance of recurrence and exploitability make it clear that no "operationality criterion" can be constructed that can guarantee that all of the generalized rules that are remembered will have positive utility in speeding up problem solving. Generalized rules are created from single cases, while recurrence can be measured only by examining the entire distribution of cases. Hence, there is no "operationality criterion" that can filter generalized rules as they are created.

4.4.3 Forgetting Rules With Low Utility

The PRODIGY system attempts to evaluate the utility of each generalized rule it learns and then "forget" rules that have negative utility. PRODIGY takes into account not only how recurrent a rule is, as suggested in Section 4.1.1, but also how manifest and how exploitable it is. The utility of a rule is expressed as the time saved by using it minus the cost of checking it. The expected time saved is the frequency with which it applies times the expected savings from using the rule in those cases to which it is applicable.

Two difficulties arise in calculating the expected utility of a generalized rule. One is the expense of gathering statistics about how frequently rules are applicable. The PRODIGY system addresses this issue by learning and evaluating utility only while solving a set of training examples. Hence, the problem solver avoids the expense of keeping statistics during normal performance. A second difficulty arises in measuring the benefit of an individual rule, when it is applicable, because the rules interact with each other. For example, the benefits from the applicability of a generalized rule that identifies a conflict set in diagnosis depend on how many suspects have already been exonerated by other generalized rules. PRODIGY finesses this second difficulty by estimating the benefits of a rule's applicability. Minton's results [Min88] demonstrate that PRODIGY estimates the utility of generalized rules well enough to improve performance by throwing out some rules. His results also suggest that better utility estimates are possible, and would improve performance even more.

Of course, filtering the generalized rules assumes that some rules will have positive utility and others will have negative utility. The analysis in this chapter points out that there are characteristics of the problem solver and the distribution of problems that will affect the utility of *all* of the generalized rules.

4.5 Conclusion

This chapter analyzed the sources of power in two common uses of EBG: generalizing successful problem solving episodes, and generalizing the explanations that search nodes are inconsistent.

There are two sources of power in generalizing successful problem solving episodes. First, the problem solver's search is biased toward search paths that have led to goal states in solving previous problems. In order to bias search effectively, however, EBG may need to be supplemented by statistical information about the frequency with which patterns of inferences are applicable. Second, the generalized rules encapsulate a pattern of operator applications, so that using a generalized rule saves the overhead cost of triggering operators and storing intermediate results that would be incurred in performing the pattern of operator applications. If, however, it is expensive to evaluate the bodies of search operators, the overhead savings from encapsulation may be outweighed by the cost of checking the generalized rules. A final caveat is that generalization of successful problem solving episodes may accelerate a search for a single solution state, but not a search for all of the solution states.

A problem solver can make two uses of generalized rules that identify search nodes as *inconsistent* with the goal state. First, the generalized rules may reduce the cost of proving that a search node is inconsistent; the generalized rule, then, is an encapsulated successful search path in a different search space, the space of proofs of inconsistency of a search node. Second, the problem solver may use the generalized rules to cut off search in the original search space. The problem solver can cut off search *below* a node that a generalized rule identifies as inconsistent, if a search node being inconsistent implies that no goal state can be reached from it. Gains from this use of EBG are often misleading, however, since a good problem solver will normally cut off the same part of the search space even without the generalized rule. The problem solver may also be able to use several generalized rules, together with simplifying assumptions (such as the single-fault assumption in circuit diagnosis), to cut off search *above* the inconsistent nodes, which the original problem solver can not do.

Chapter 5

Similar = Same Fault Hypothesis

Chapters 2 and 3 analyzed one dimension of similarity in detail. The next two chapters explore alternative definitions of similarity and the learning methods that arise from them. In this chapter, we extend the diagnostic task to include the identification of specific misbehaviors for components. Then, we define two sets of observations as similar if the same misbehavior of the same component can explain the symptoms of both. For example, the observations in both of the cases presented in the introduction (repeated as Figure 5.1) can be explained by the first bit of M1's output being *stuck low*. In the first case, M1 outputs 4 as the product (instead of 6), which A1 adds to the 6 predicted at Y to produce 10 at F, which agrees with the observation. In the second case, M1 outputs 0 (instead of 2), which A1 adds to the 5 predicted at Y to produce 5 at F, which agrees with the observation.

One way to make manifest such a similarity between cases is to generalize the reasoning process used to check the consistency of a fault hypothesis in one case, then check if the generalized rule is applicable to the other case. This is another use of the EBG technology for generalizing patterns of inferences. We also propose a technique called *lifting* for creating generalized rules that check the consistency of fault hypotheses regardless of the reasoning process used by the diagnostic program. A very interesting, but as yet unimplemented, idea is to use a design verification to guide simplification of the preconditions in the lifted rules, to make them more efficient.

5.1 Fault Hypotheses

Thus far in this thesis, the troubleshooter has not considered component failure modes. It has been concerned only with identifying the components for which a misbehavior of *some* kind could account for all of the symptoms. Some kinds of component misbehavior are more plausible than others. For example, if adder A1 is implemented as a single TTL chip, it would fail in predictable ways (e.g. pins coming

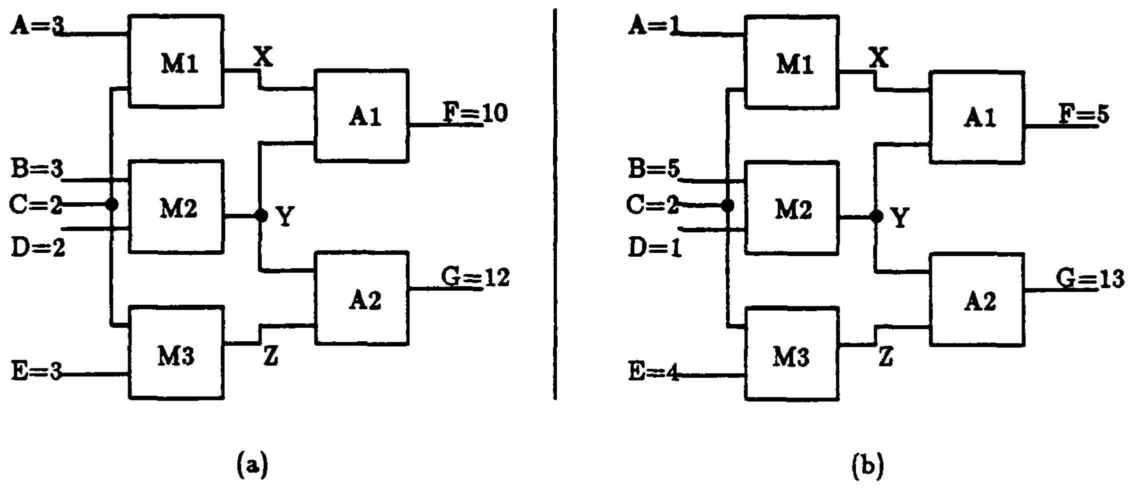


Figure 5.1: The Polybox Circuit. Both of the sets of observations shown are consistent with the first bit of A1's upper input being stuck low.

loose) that yield predictable misbehaviors (stuck-at faults).

We now extend the diagnostic engine's task to finding specific fault hypotheses for the candidate components that it finds. The diagnostic engine is provided with a list of the common modes of failure of each of the components in the device it diagnoses. After it finds the candidate set, it proposes as a fault hypothesis each of the modes of failure that it knows about for each candidate component and checks whether each hypothesis can explain all of the device behavior. The program outputs the fault hypotheses that are confirmed.

5.2 Generalizing Fault Envisionments

Given the observations of Figure 5.1a), M1 and A1 are the single-fault candidates. The program then looks up the known potential misbehaviors for M1 and A1 and checks each for consistency with the observations. One possibility is that the first bit of M1's output is stuck-at 0. The program performs a simulation, referred to as a *fault envisionment*, to determine whether this is a consistent fault hypothesis. It propagates the inputs through the components, computing the hypothesized *misbehavior* for M1 and correct behaviors for the other components (see Figure 5.2.) This yields values at the circuit outputs. In this case, the predicted outputs are consistent with the observed outputs, so the fault hypothesis is consistent.

Another possibility is that the zeroth bit of M1's output is stuck-at 1. In that case, a fault envisionment would predict the value 7 at X, whence 13 at F, which contradicts the observed value of 10. Hence, that fault hypothesis is eliminated.

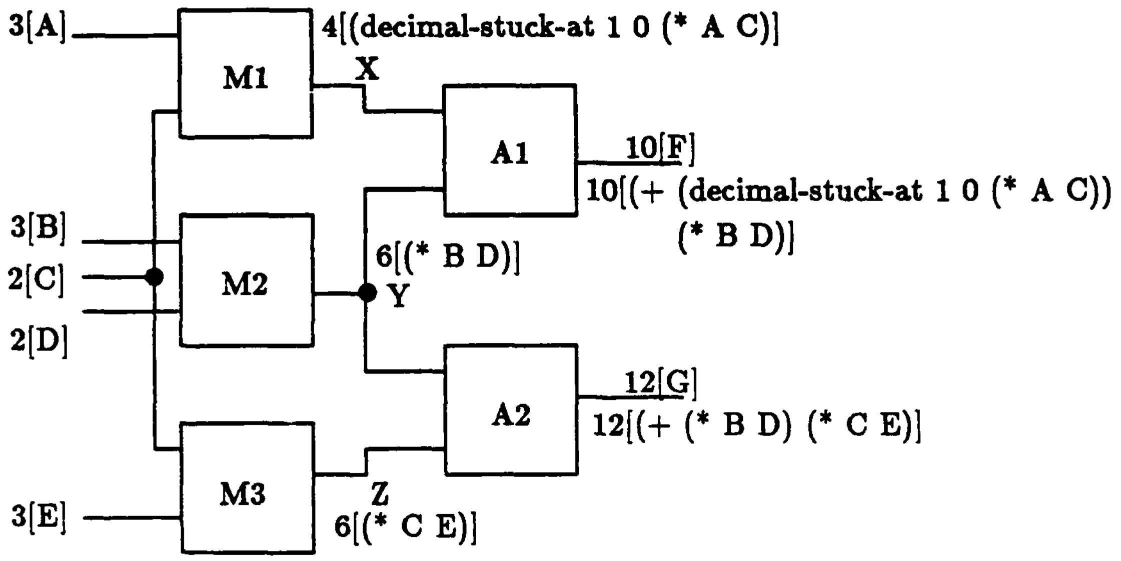


Figure 5.2: Envisioning the hypothesis that the first bit of M1's output is stuck-at 0.

By encapsulating particular fault envisionments, EBG can find sufficient conditions for inferring that a fault hypothesis is consistent or sufficient conditions for inferring that it is inconsistent with the observations. The expressions in brackets in Figure 5.2 illustrate generalizing the envisionment of the first bit of M1's output being stuck-at 0. The resulting generalized rule is:[1]

```
R6: IF (AND (= ?F (+ (decimal-stuck-at 1 0 (* ?A ?C))
                     (* ?B ?D)))
            (= ?G (+ (* ?B ?D) (* ?C ?E))))
    THEN (fault-hypothesis '(decimal-stuck-at 1 0 (OUTPUT M1)))
```

The envisionment that eliminated the hypothesis that the zeroth bit of M1's output is stuck-at 1 can also be generalized. It yields the rule:

```
R7: IF (NOT (= ?F (+ (decimal-stuck-at 1 0 (* ?A ?C))
                     (* ?B ?D))))
    THEN (not-fault-hypothesis '(decimal-stuck-at 1 0 (OUTPUT M1)))
```

Note that encapsulations of envisionments give only sufficient conditions for the confirmation or elimination of a fault hypothesis. The reason is that any particular fault envisionment may use only part of the behavior of some components. Hence, even if the inferences in a particular envisionment confirming a fault hypothesis do not apply to a new case, the program cannot eliminate the fault hypothesis. As it turns

[1]**decimal-stuck-at** is a function that takes three arguments, a bit to stick, the value it's stuck-at (0 or 1), and a decimal number. The bit can be any number from 0 up to the number of bits in the decimal number.

out, R6 does give necessary and sufficient conditions for checking the hypothesis that the first bit of M1's output is stuck-at 0, but that can not be counted on in general.

5.2.1 Utility of EBG on Envisionments

The program can use the generalized rules to jump to the conclusion that a fault hypothesis is consistent (or inconsistent, depending on the rule) without having to perform the fault envisionment. For either kind of generalized rule, this is an example of using EBG to encapsulate successful patterns of behavior rule firings, as in Section 4.1. We draw on the analysis from that section to characterize when performance will improve.

Search Reduction

It is not always necessary to propagate values through every device component in order to prove the consistency (inconsistency) of a particular fault hypothesis. Generalized rules that identify fault hypotheses as consistent (inconsistent) can bias the search involved toward patterns of value propagations that were useful in solving past cases. As discussed in Section 4.1.1, this search bias will be effective to the extent that many value propagations are *never* used in proving the consistency (inconsistency) of a particular fault hypothesis. Depending on the distribution of cases presented to the problem solver, the search bias may be more or less effective.

In addition, the topology of the device may be such that some components will never contribute to proving a particular fault hypothesis *inconsistent*, regardless of the actual distribution of cases presented to the problem solver. For example, a component X may contribute to only one or a few circuit outputs. Hence, only components that can contribute to the same outputs X contributes to will ever be useful in proving that a fault hypothesis about X is inconsistent. Generalized rules that prove the inconsistency of a fault hypothesis about component X will all ignore the "irrelevant" components, regardless of the distribution of cases. In performing a fault envisionment for X, however, the original problem solver would not know to ignore those "irrelevant" components, so the generalized rules reduce the amount of search.

Effect of Encapsulation

The second source of power from generalizing successful patterns of inferences lies in checking the weakest preconditions and jumping to conclusions rather than computing all of the intermediate steps. Since checking a generalized rule requires paying all of the behavior costs of the encapsulated rule firings, and some rule firings will be encapsulated in several generalized rules, the benefits depend on the relative costs of evaluating the bodies of behavior rules versus the overhead costs of triggering behavior rules (binding variables in their left-hand sides) and storing their results. The cost of evaluating the body of a behavior rule depends on the complexity of the component. The overhead costs should be low when doing fault envisionment, as is argued below. The cost of triggering behavior rules with a constraint network is low. It is not necessary to keep track of dependencies, because the program is only interested in finding out whether a fault hypothesis is consistent, so recording a new value should be inexpensive. Remember that recording dependencies was a major cost of asserting a new value in Section 2.1, where the program needed the dependency structure in order to find the components supporting derivations of contradictory values. Hence, the effect of encapsulation in speeding up performance of fault envisionment should be minor if the component behaviors are inexpensive, and may be negative if the component behaviors are expensive.

Summary

The program may be able to use generalized fault envisionments to reduce the cost of checking the consistency of a fault hypothesis. It can use generalized rules to bias the search for a proof of the consistency (or inconsistency) of a fault hypothesis if some components never contribute to proving the consistency (or inconsistency) of a fault hypothesis. The overhead saving from checking generalized rules rather than propagating values will not be significant, because the program need not keep track of dependencies during fault envisionments. The overhead saving may be overshadowed by the cost of computing some components' behavior repeatedly in several generalized rules, especially if the components have complex behavior.

5.3 Lifting Fault Hypotheses

It is possible to construct both necessary and sufficient conditions for recognizing whether or not a fault hypothesis is consistent with observations. We call the process *lifting*. A fault hypothesis for a *component* is lifted to a fault hypothesis for the *device* using a symbolic fault envisionment. As illustrated in Figure 5.3, a symbolic fault envisionment is an envisionment in which variables are used for the inputs. The symbolic simulation can be packaged into a generalized rule for checking the consistency of a given fault hypothesis in future cases. The generalized rule simply composes the behaviors (and misbehaviors) of the components:

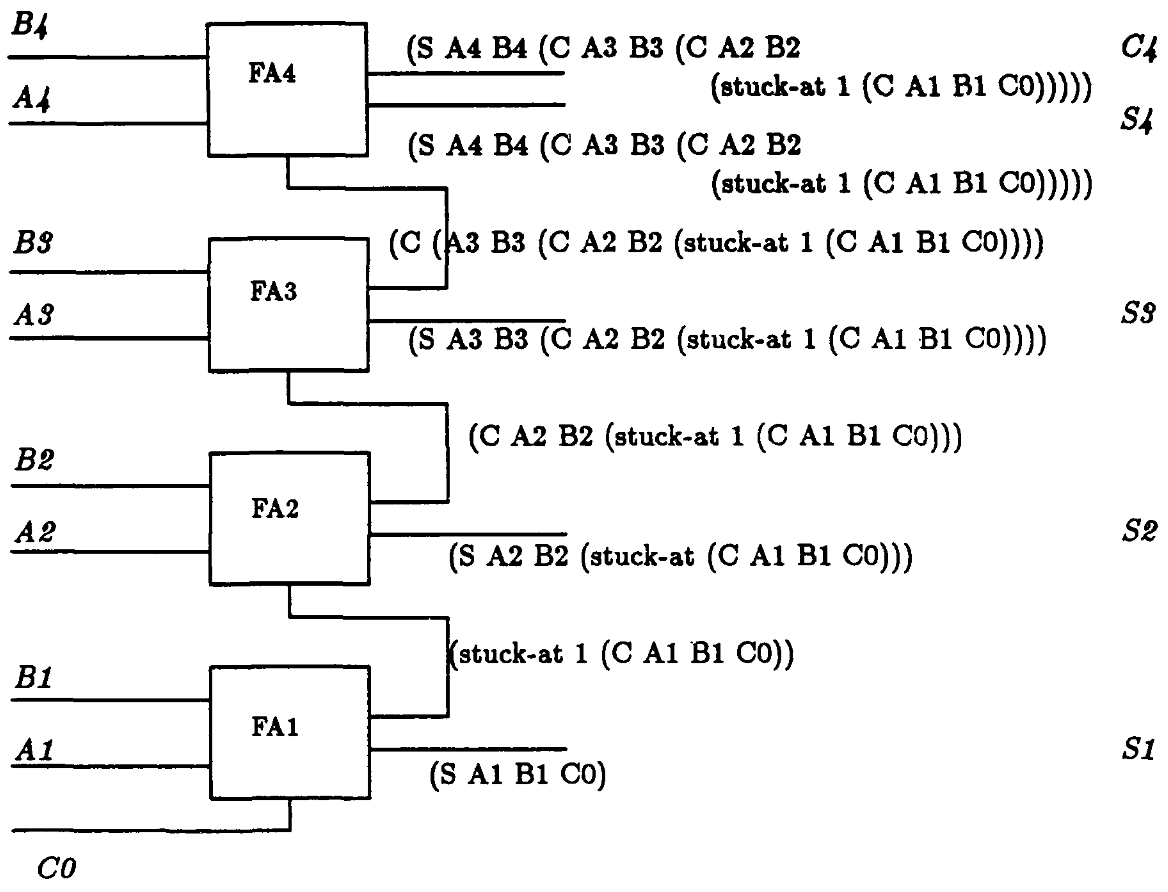

B4
A4
FA4
(S A4 B4 (C A3 B3 (C A2 B2 (stuck-at 1 (C A1 B1 C0)))))
C4
(S A4 B4 (C A3 B3 (C A2 B2 (stuck-at 1 (C A1 B1 C0)))))
S4
B3
A3
FA3
(C (A3 B3 (C A2 B2 (stuck-at 1 (C A1 B1 C0))))
(S A3 B3 (C A2 B2 (stuck-at 1 (C A1 B1 C0))))
S3
B2
A2
FA2
(C A2 B2 (stuck-at 1 (C A1 B1 C0)))
(S A2 B2 (stuck-at (C A1 B1 C0)))
S2
B1
A1
FA1
(stuck-at 1 (C A1 B1 C0))
(S A1 B1 C0)
S1
C0

Figure 5.3: A carry-chain adder composed of four full adders. Lifting the fault hypothesis that FA_1's carry-bit is stuck-at 1. S refers to the sum-bit (the low-order bit of the sum) of three single-bit arguments and C refers to the carry-bit (the high-order bit).

```
R8:
IF (AND (= ?S1 (S ?A1 ?B1 ?CO))
       (= ?S2 (S ?A2 ?B2 (STUCK-AT 1 (C ?A1 ?B1 ?CO)))
       (= ?S3 (S ?A3 ?B3
                 (C ?A2 ?B2 (STUCK-AT 1 (C ?A1 ?B1 ?CO)))))
       (= ?S4 (S ?A4 ?B4
                 (C ?A3 ?B3
                       (C ?A2 ?B2
                             (STUCK-AT 1 (C ?A1 ?B1 ?CO))))))
       (= ?C4 (C ?A4 ?B4
                 (C ?A3 ?B3
                       (C ?A2 ?B2
                             (STUCK-AT 1 (C ?A1 ?B1 ?CO)))))))
 THEN (FAULT-HYPOTHESIS '(STUCK-AT 1 (C-OUT FA1)))
```

While this can be viewed as an application of EBG, here the reasoning process that is generalized is not one that is employed by the original problem solver. Our previous use of EBG has been to encapsulate only that part of the components' behavior that was actually used to deduce some conclusion, and not the most general behavior of all of the components in the circuit. The symbolic fault simulation contains all of the behavior of the circuit components. As a result, encapsulating it yields necessary and sufficient conditions for the observations to be consistent with a particular fault hypothesis.

There are difficult issues involved in symbolic simulation that are beyond the scope of this thesis. First, the symbolic simulation may involve iterative behavior, so that the straightforward simulation may not end. Weld and others [Wel86, SD87, CMB88] addressed the issue of noticing and generalizing iterative behavior in simulations. Generalizations of their techniques may apply to symbolic simulations. Second, components that compute conditional outputs (e.g. multiplexers) may make the composed behavior expressions prohibitively large.

5.3.1 Utility of Lifting

The analysis of the utility of lifted rules is somewhat different than previous utility analyses because a lifted rule provides *necessary* as well as sufficient conditions for deciding the consistency of a fault hypothesis. There can be only one lifted rule for a given fault hypothesis and the only question is whether checking the lifted rule takes more or less time than performing a fault envisionment.

A lifted rule will increase the number of component behaviors that are simulated, but avoids the overhead costs of running behavior rules. Unlike the generalizations of fault envisionments, a lifted rule encapsulates all of the behavior of the circuit components. Any particular fault envisionment may use the behavior of only some of the components. Hence, at least as much, and possibly more, component behavior is simulated in checking a lifted rule as would be simulated during a fault envisionment. On the other hand, checking a lifted rule avoids the matching of component rule preconditions and the storage of intermediate results. To repeat a frequent theme in this thesis: the overall utility of a lifted rule depends on the relative costs of the component behaviors versus the overhead costs of the rule system.

The utility of a lifted rule may be increased by simplifying the expressions in its preconditions. In general, expression simplification is an intractable problem. Some guidance may be available, however, from a design verification, a proof that the behavior specification for a device is met by its implementation. Design verifications might plausibly be generated during the design process, either by human designers or by a computer program. Simplifying the expressions in the preconditions of the lifted rules will reduce the cost of checking them.

Design verifications may make the problem tractable by providing some guidance to the process of expression simplification. The chief heuristic is:

- If the behavior of only one component changes, a proof that the circuit implements its designed behavior may go through almost unchanged.

A hand demonstration of this idea is given below. Automating this process is a very interesting problem for future research, since real cases are likely to be much more difficult than the example below.

Consider the carry chain adder in Figure 5.3. FA_1 through FA_4 compute the sum of three one-bit numbers. The component behavior rules compute $(Sabc)$, the sum-bit of inputs a, b, and c, and $(Cabc)$, the carry-bit. The circuit adds two decimal numbers between 0 and 15, plus a single-bit carry-in. The proof that the circuit meets the design specification relies on two abstractions: the conversion from binary to decimal representation of integers and the positional representation of binary numbers on the wires of the circuit. These abstractions are reflected in the following equations, which are used repeatedly in the design verification.

$$\begin{aligned} A &= a_1 + 2a_2 + 4a_3 + 8a_4 \\ B &= b_1 + 2b_2 + 4b_3 + 8b_4 \\ Output &= s_1 + 2s_2 + 4s_3 + 8s_4 + 16c_4 \\ a + b + c &= (S\, abc) + 2(C\, abc) \end{aligned}$$

A proof that the adder is implemented by its substructure is shown in Figure 5.4. We assume that such a proof would be provided as an input to a diagnostic program. The first two lines of the derivation give the output as a composition of the component behaviors. This composition of behaviors is then simplified, using the behavioral abstraction equations above.

Now consider the effect on this proof of assuming that FA_1 is computing an incorrect carry-bit. This might be the case if the carry-bit were stuck-at 1. More generally, suppose that the carry bit computed is some function g of the inputs. Thus, $FA_1(a_1b_1c_0) = (S\ a_1b_1c_1) + 2(g\ a_1b_1c_1)$. Intuitively, the adder circuit will now add its inputs, but with an error term corresponding to twice the error on the carry-bit of FA_1. In Figure 5.5 we see that, by expressing the faulty behavior as the correct behavior plus an error term, essentially the same proof goes through, except that an error term is left over.

Note that this technique will work even with multiple faults, although there will be correspondingly more error terms. The result of this replayed proof would be the following, simpler version of R8:

$$\begin{aligned}
Output &= s_1 + 2s_2 + 4s_3 + 8s_4 + 16c_4 \\
&= (S\, a_1b_1c_0) + 2(S\, a_2b_2(C\, a_1b_1c_0)) \\
&\quad +4(S\, a_3b_3(C\, a_2b_2(C\, a_1b_1c_0))) \\
&\quad +8(S\, a_4b_4(C\, a_3b_3(C\, a_2b_2(C\, a_1b_1c_0)))) \\
&\quad +16(C\, a_4b_4(C\, a_3b_3(C\, a_2b_2(C\, a_1b_1c_0)))) \\
&= (S\, a_1b_1c_0) + 2(S\, a_2b_2(C\, a_1b_1c_0)) \\
&\quad +4(S\, a_3b_3(C\, a_2b_2(C\, a_1b_1c_0))) \\
&\quad +8(a_4 + b_4 + (C\, a_3b_3(C\, a_2b_2(C\, a_1b_1c_0)))) \\
&= (S\, a_1b_1c_0) + 2(S\, a_2b_2(C\, a_1b_1c_0)) \\
&\quad +4(a_3 + b_3 + (C\, a_2b_2(C\, a_1b_1c_0))) \\
&\quad +8(a_4 + b_4) \\
&= (S\, a_1b_1c_0) + 2(a_2 + b_2 + (C\, a_1b_1c_0)) \\
&\quad +4(a_3 + b_3) + 8(a_4 + b_4) \\
&= c_0 + (a_1 + b_1) + 2(a_2 + b_2) + 4(a_3 + b_3) + 8(a_4 + b_4) \\
&= c_0 + A + B
\end{aligned}$$

Figure 5.4: A design verification for the carry-chain adder.

```
R8':
IF (AND (= ?Output
           (+ ?CO ?A ?B
              (* 2 (- (g ?A1 ?B1 ?CO) (C ?A1 ?B1 ?CO)))))
THEN (FAULT-HYPOTHESIS '(STUCK-AT 1 (CARRY-OUT FA1)))
```

Even though R8' encapsulates the behavior of every component, evaluating the simplified expressions may be more efficient than simulating the components individually in a fault simulation.

5.3.2 Summary

Lifted rules may be more efficient than fault simulation for checking the consistency of fault hypotheses, if expressions in the lifted rules' preconditions can be simplified. An interesting direction for future research would be to automate the use of design verifications to guide the simplification of lifted rules' preconditions.

$$\begin{aligned}
\mathit{Output} &= s_1 + 2s_2 + 4s_3 + 8s_4 + 16c_4 \\
&= (S\ a_1b_1c_0) + 2(S\ a_2b_2(g\ a_1b_1c_0)) \\
&\quad +4(S\ a_3b_3(C\ a_2b_2(g\ a_1b_1c_0))) \\
&\quad +8(S\ a_4b_4(C\ a_3b_3(C\ a_2b_2(g\ a_1b_1c_0)))) \\
&\quad +16(C\ a_4b_4(C\ a_3b_3(C\ a_2b_2(g\ a_1b_1c_0)))) \\
&= (S\ a_1b_1c_0) + 2(S\ a_2b_2(g\ a_1b_1c_0)) \\
&\quad +4(S\ a_3b_3(C\ a_2b_2(g\ a_1b_1c_0))) \\
&\quad +8(a_4 + b_4 + (C\ a_3b_3(C\ a_2b_2(g\ a_1b_1c_0)))) \\
&= (S\ a_1b_1c_0) + 2(S\ a_2b_2(g\ a_1b_1c_0)) \\
&\quad +4(a_3 + b_3 + (C\ a_2b_2(g\ a_1b_1c_0))) \\
&\quad +8(a_4 + b_4) \\
&= (S\ a_1b_1c_0) + 2(a_2 + b_2 + (g\ a_1b_1c_0)) \\
&\quad +4(a_3 + b_3) + 8(a_4 + b_4) \\
&= (S\ a_1b_1c_0) \\
&\quad +2(a_2 + b_2 + (C\ a_1b_1c_0) + (g\ a_1b_1c_0) - (C\ a_1b_1c_0)) \\
&\quad +4(a_3 + b_3) + 8(a_4 + b_4) \\
&= c_0 + (a_1 + b_1) + 2(a_2 + b_2) + 2((g\ a_1b_1c_0) - (C\ a_1b_1c_0))) \\
&\quad +4(a_3 + b_3) + 8(a_4 + b_4) \\
&= c_0 + A + B + 2((g\ a_1b_1c_0) - (C\ a_1b_1c_0))
\end{aligned}$$

Figure 5.5: Using the design verification to guide simplification of the preconditions of R8.

5.4 Exploitability

Constraint suspension sometimes provides enough information about how a component must be misbehaving that the program can identify the consistent fault hypotheses for the component without resorting to fault envisionment. For example, in performing constraint suspension on M1 while diagnosing the polybox circuit in Figure 5.1a), the program would predict the value 4 at M1's output from 10 at F and 6 at Y. Given values for M1's inputs and outputs, the program can check the consistency of the fault hypothesis that M1's zeroth output bit is stuck high (it is consistent) without resorting to fault envisionment for the whole circuit. In such situations, the program should not check any generalized rules constructed from fault envisionments or lifted rules.

There are also situations, however, in which it is necessary to use fault envisionment or generalized rules to check fault hypotheses for a component given only the

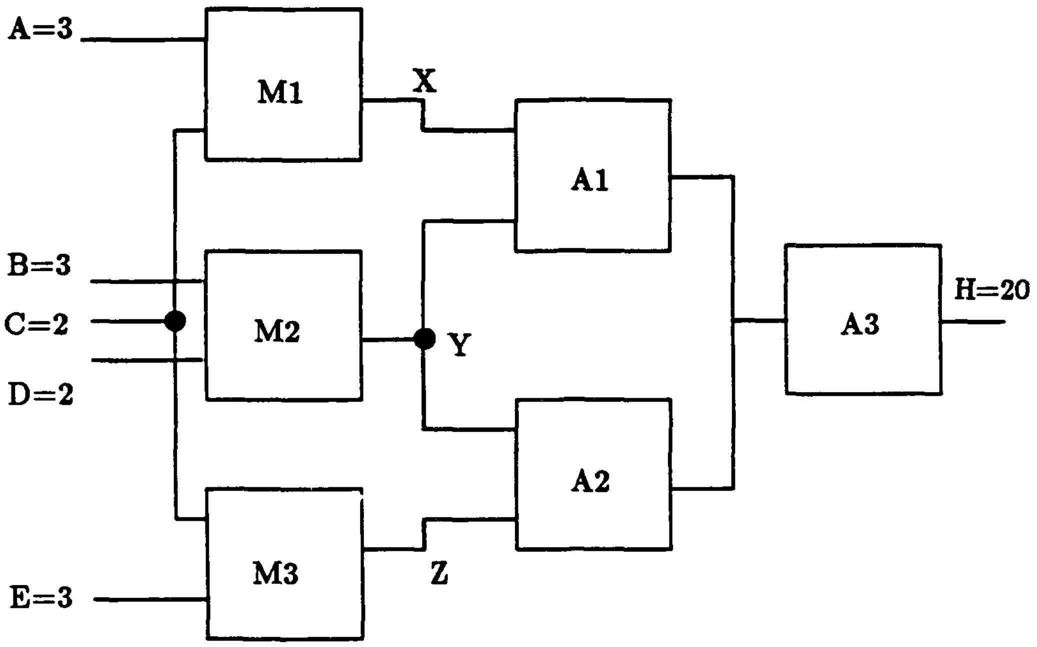


Figure 5.6: Envisionments are needed to check the consistency of fault hypotheses for M2.

device's inputs and outputs. Figure 5.6 demonstrates a simple example of a circuit in which constraint suspension is unable to predict inputs and outputs for some components (the presence of reconvergent fanout often leads to failure of local propagation). With M2 turned off, no value is predicted at X, even though M2 must produce the value 4 in order to account for the device behavior.[2]

During candidate generation, constraint suspension is performed on every suspect that is not exonerated. Thus, it is easy for the program to identify when the generalized (or lifted) rules should be checked. The generalized rules are checked only when constraint suspension fails to identify how the candidate must be misbehaving in order to account for the device observations.

5.5 Related Work

Pazzani's ACES program [Paz86] for diagnosing failures in the attitude control system of a satellite used EBG to generalize what we call fault envisionments that proved the inconsistency of fault hypotheses. Pazzani presented empirical evidence demonstrating that EBG improved performance on a few examples, but did not present an analysis of the source of that speedup. As discussed in Section 5.2, the

[2]Note that if H were 21, M2 would not be exonerated during candidate generation, even though it would have to output 4.5 to account for the device behavior.

key factor, if performance is to improve on a large set of examples, is that the behavior of some components never contribute to proving the inconsistency of certain fault hypotheses. Hence, even though there may be more than one generalized rule for identifying a fault hypothesis as inconsistent, the behavior of the irrelevant components will not be encapsulated in any of those generalized rules.

The idea of composing component behaviors and then simplifying expressions is not new. Weise's Silica Pithecus [Wei86] and Barrow's VERIFY [Bar84] both do so in order to verify device designs. Hall, Lathrop, and Kirk [HLK87] use the idea to turn a structural and behavioral description for a device into a faster simulator. In all three cases, brute force or ad hoc heuristic methods were used to guide expression simplification. What is novel in our work is the idea of replaying a design verification to guide the simplification of expressions.

5.6 Conclusion

This chapter defined sets of observations for a circuit to be similar if they were consistent with the same fault hypothesis. We presented two methods for constructing generalized rules that could check for such similarities. The first was to generalize the reasoning process used in fault envisionments. The second was to lift a description of a component misbehavior into a description of a device misbehavior, using symbolic simulation. We suggested, but did not implement, the use of design verifications to guide the simplification of the lifted behavior expressions.

Chapter 6

Extensions: Relaxing Similarity Definitions

This chapter suggests directions for future research on ways to relax the two notions of similarity described in previous chapters. First, in Chapter 2, we defined two sets of observations for the same device to be similar if the same derivation of contradictory values was applicable to both. That is, two sets of observations were similar if the observations could be propagated through exactly the same components, leading to a contradiction at exactly the same location. In this chapter we go further, and define notions of similarity for patterns of inferences, which in turn allows us to define sets of observations as similar if merely *similar* derivations of contradictory values are applicable. Second, in Chapter 5, we defined two cases to be similar if they were consistent with exactly the *same* fault hypothesis (i.e., misbehavior for a particular component.) In this chapter, we define notions of similarity for fault hypotheses, which allows us to define two cases as similar if they are consistent with merely *similar* fault hypothese.

Figure 6.1 summarizes the recursive nature of the similarity definitions in this thesis. Defining similarity for sets of observations is reduced to defining similarity for fault hypotheses (or patterns of inferences), and so on. As we will see in this chapter, the recursive definitions must bottom out either in a strict equality test, or in a primitive definition of similarity that is provided to the program.

6.1 Similarities Between Patterns of Inferences

As described above, we can define similarity of observations in terms of similarity of patterns of inferences (derivations): two sets of observations are similar if similar patterns of inferences are applicable to them. We now have to define similarity for patterns of inferences, which we do in two ways. First, we use information about equivalent roles that different components play to associate sequences of value prop-

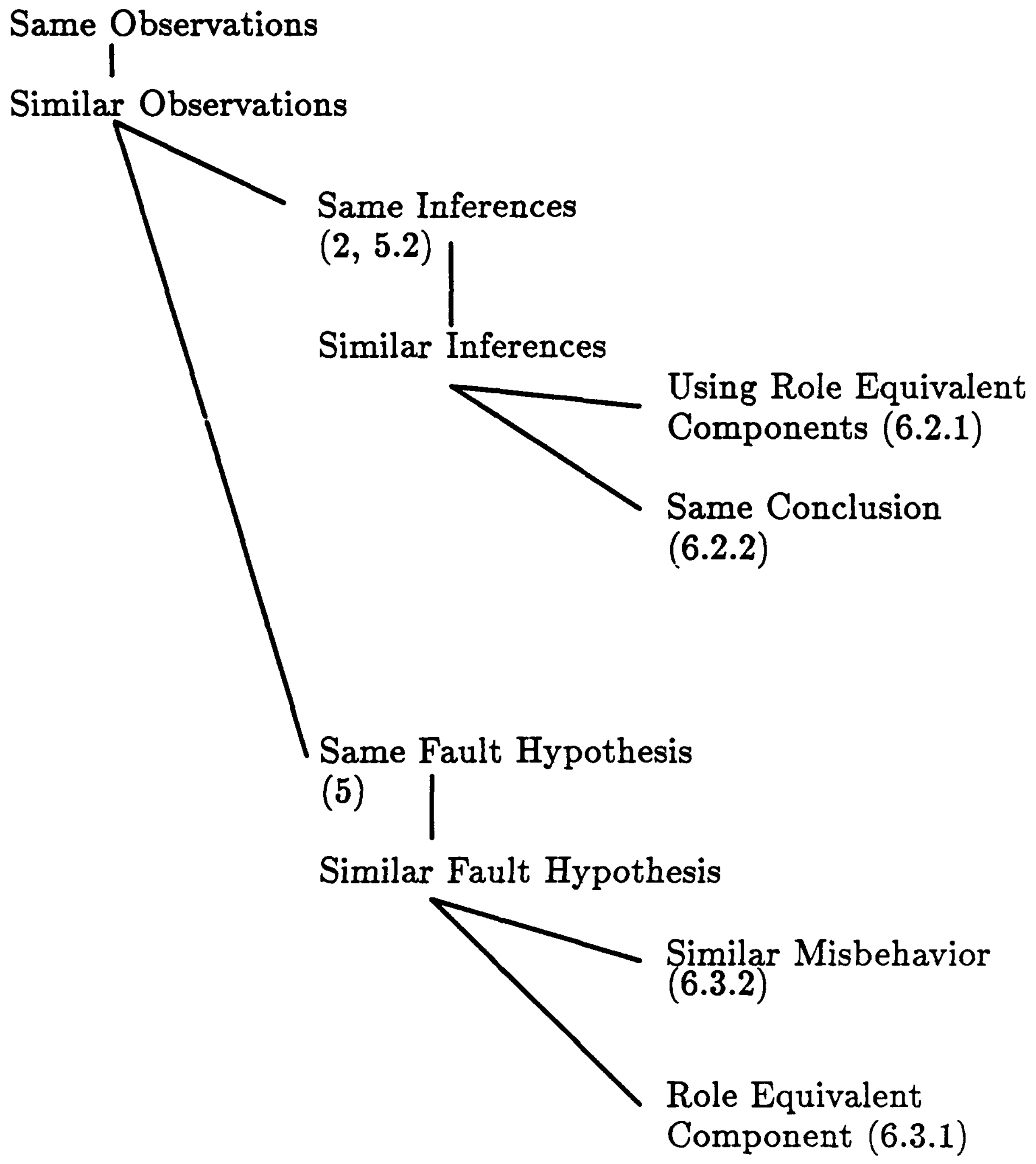


Figure 6.1: The definitions of similarity proposed in this thesis. Moving to the right and down indicates definitions that classify more cases as similar.

agations that occur in different places in the circuit. Second, we define two patterns to be similar if they lead to the same conclusion.

6.1.1 Role Equivalent Conflict Sets

Often, many components in a device perform the same role. For example, M1 in polybox is performing the same role as M3, similarly for A1 and A2. In the carry-lookahead adder each of the XOR gates is computing a sum bit. We assume the program is given information about role equivalences of components as part of the device description. An even more ambitious project would be to have the program deduce the role equivalences from the structure and behavior description of the device.

We use role equivalences to define two derivations as similar if they use "equivalent" components. This leads to the definition of two sets of observations as similar if derivations of contradictory values using "equivalent" components can be instantiated in both. Using this notion, the learning program could construct and generalize derivations of additional conflict sets that are analogous to those derived in diagnosing a specific case.

An easy example of this idea occurs in diagnosis of the polybox circuit. Figure 2.4 showed the derivation of a contradiction at output F, using components M1, M2 and A1, from which EBG generated the rule:

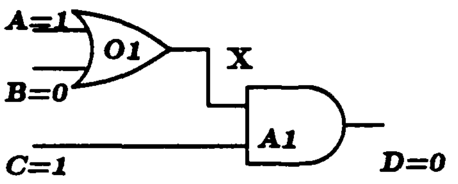
```
R1: IF (NOT (= ?F (+ (* ?A ?C) (* ?B ?D))))
    THEN (CONFLICT-SET '(M1 M2 A1))
```

Suppose the program is given the role equivalence that M1 plays the same role as M3 and A1 plays the same role as A2. Simply by substituting "equivalent" components for equivalents in the derivation of the contradiction at output F, then generalizing the new derivation, the program could generate the rule:

```
R10: IF (NOT (= ?G (+ (* ?C ?E) (* ?B ?D))))
     THEN (CONFLICT-SET '(M3 M2 A2))
```

Constructing an analogous conflict set for the adder circuit is more difficult. A derivation of a contradiction on the first output bit should be analogous to a derivation of a contradiction on any other bit. A contradiction on the third output bit, however, can depend on more inputs than a contradiction at the first bit. Constructing the analogous derivation of a contradiction would require more effort than simply substituting "equivalent" components for equivalents in the original derivation. We leave this as a problem for future research.

It is interesting to note that while constructing and generalizing analogous derivations of contradictions can speed up the learning process, it would not affect performance in the long run. A rule like R10 above can provide some savings the first time



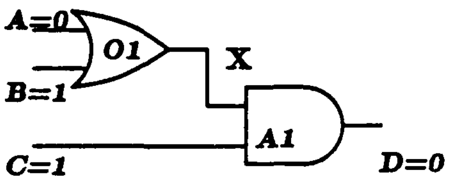


Figure 6.2: The two examples allow different derivations that yield the same conflict set, (O1 A1).

it is applicable. However, it is only in diagnosing that first case to which it applies that any benefit is gained, because R10 would be constructed during the diagnosis of that first case if it had it not been constructed previously. Hence, constructing and generalizing analogous derivations of conflict sets is an interesting idea, but not a useful one for performance learning.

6.1.2 Same Conclusion; Different Reasoning

Another way to define two patterns of inferences as similar is if they both lead to the same conclusion. This leads to defining two sets of observations as similar if the diagnostic engine can reach the same conclusion from both, perhaps by a different line of reasoning. This weaker notion of similarity leads us to try to construct a single generalized rule that is applicable when any of the patterns of inferences leading to a particular conclusion are applicable. There are two potential efficiency advantages to combining the preconditions of several generalized rules that have the same conclusions into a single rule. First, it may be possible to collapse the preconditions, making it more efficient to check the single combined rule than all of the individual ones. Second, the program may be able to conclude that it has found all of the possible ways to derive a particular conclusion, so that the combined rule provides necessary and sufficient conditions for reaching that conclusion, rather than just sufficient conditions.

Alternate Derivations of the Same Conclusion

Figure 6.2 shows a situation where it is possible to have two derivations of contradictions that yield the same conflict set. The value 1 at X can be predicted either from a 1 at A or from a 1 at B. Consider now the generalizations resulting from two cases. The first case has A=1, B=0, C=1, and D=0. O1 and A1 together predict 1 at D, which is a contradiction. The generalization is:

```
R11: IF (AND (= ?A 1)
             (NOT (= ?D (AND 1 ?C)))
     THEN (CONFLICT-SET '(O1 A1))
```

The second case has A=0, B=1, C=1, and D=0. Again there is a contradiction at D. The generalization is:

```
R12: IF (AND (= ?B 1)
             (NOT (= ?D (AND 1 ?C)))
     THEN (CONFLICT-SET '(O1 A1))
```

Collapsing Preconditions The above two rules can be combined into a single disjunctive rule, whose preconditions can then be collapsed. If both rules are checked independently, the predicate (NOT (= ?D (AND 1 ?C))) will be evaluated twice. The following combined rule is more efficient to check:

```
R13: IF (AND (OR (= ?A 1) (= ?B 1))
             (NOT (= ?D (AND 1 ?C))))
     THEN (CONFLICT-SET '(O1 A1))
```

Hence, by combining the two rules and collapsing the preconditions, the cost of checking all of the generalized rules during diagnosis can be reduced, thus improving the overall performance.

Necessary and Sufficient Conditions

The problem with a rule that gives only sufficient conditions for reaching its conclusion is that nothing can be concluded from the failure of the rule to apply. If a problem solver were to know that a rule has necessary and sufficient conditions for reaching its conclusions, the problem solver would be able to exploit negative results from checking the rule as well as positive results. This section discusses how the problem solver could exploit necessary and sufficient conditions for constructing conflict sets and also how the program might be able to generate them.

There are two ways that the augmented diagnostic program of Chapter 2 could be improved if it knew that it had necessary and sufficient conditions for some conflict sets. First, the inapplicability of a rule with necessary and sufficient conditions for identifying a conflict set implies that no rule for constructing a conflict set that is a subset of the original will ever succeed. If it is not possible to derive a contradiction using M1, M2, and A1, it certainly will not be possible to derive a contradiction using only M1 and A1. Thus, having necessary conditions for some rules would enable the program to test those rules first; if they fail, the program need not consider rules that have smaller conflict sets as their conclusions.

Second, if the program has necessary and sufficient conditions for *all* of the possible conflict sets, it is no longer necessary to perform constraint suspension on the suspects that are still left after having checked the rules from the conflict set library. This improvement would make the algorithm the same as the optimistic algorithm of Section 3.9, while still guaranteeing the most specific diagnoses possible.

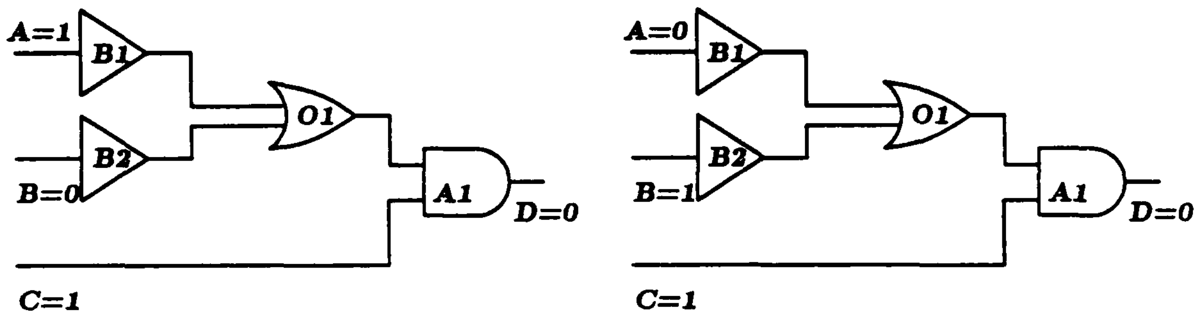


Figure 6.3: B1 and B2 are buffers. The first example allows a derivation that yields the conflict set (B1 O1 A1). The second example allows a different derivation that yields a different conflict set, (B2 O1 A1).

While a program may often have necessary and sufficient conditions for a particular conflict set, it is difficult for the program to *know* that it has necessary and sufficient conditions. There is hope, however, because there can be more than one derivation of a conflict set only in unusual circumstances. Consider the characteristics of the situation in Figure 6.2 that allowed there to be more than one derivation of conflict set (O1 A1). First, there are behavior rules for O1 that use only one of the inputs to predict the output. That makes it possible for one derivation to use one input of O1 and another derivation to use a different input. Second, each of O1's inputs is connected to the same set of components (in this case, none).

As a contrasting example, consider Figure 6.3, in which the two inputs of O1 are connected to two different buffers, B1 and B2. As before, the two sets of observations allow two different derivations of contradictions at D, and EBG constructs a pair of rules with the same left-sides as R11 and R12. In this example, however, the conflict sets constructed would now include B1 in the first case and B2 in the second, so the two derivations yield different conflict sets.

If the program can prove that it has seen all of the possible derivations of a particular conflict set, it will know that it has necessary and sufficient conditions for that conflict set. We hypothesize that the only way that there can be more than one derivation of the same conflict set is a situation where both inputs to a component are either inputs to the circuit, as in Figure 6.2, or are connected to the same component. A program may be able to use this hypothesis to prove that it has found all of the possible derivations of a particular conflict set. Future research is required, however, to check the validity of the hypothesis.

6.2 Similarities Between Fault Hypotheses

As described in the introduction to this chapter, we can define similarity of sets of observations in terms of similarities between fault hypotheses. A fault hypothesis is a proposed misbehavior for a component. We define similarities between fault hypotheses in two ways. First, they can be similar if they propose the same misbehavior for two components playing the same role. Second, they can be similar if they propose similar misbehaviors for the same components.

6.2.1 Same Misbehavior; Role Equivalent Component

We first consider two fault hypotheses to be similar if they propose the same misbehavior for components playing equivalent roles in the circuit. This leads us to define two sets of observations as similar if they are both consistent with a particular misbehavior on some component playing a particular role.

In order to recognize such a similarity, we propose that the fault lifting technique of Section 5.3 be parameterized. The lifted fault hypotheses for different components playing the same role should not be very different. For example, the behavior of the carry-chain adder when any of the carry-bits of the component adders is stuck-at one can be expressed as correct addition plus an error term. Ideally, one rule could be constructed with a parameterized error term; the rule would check simultaneously for the fault hypothesis on any of the components playing the same role. Checking the parameterized rule would be more efficient than checking a separate rule for each component playing the same role.

6.2.2 Similar Misbehavior; Same Component

Another way to define fault hypotheses as similar is if they propose similar misbehaviors for the same components. Here, similarity for two misbehaviors means that there is some common generalization of the two. We assume that the diagnostic engine is given a hierarchy of misbehavior descriptions (sometimes called fault models). For example `unspecified-stuck-at` is more general than both `stuck-at-1` and `stuck-at-0`. This leads to defining two sets of observations as similar if they are both consistent with the same general misbehavior for the same component. In order to recognize such similarities, the program could create a generalized rule with the lifting technique of Section 5.3, using the generalized misbehavior.

6.3 Summary

In this chapter we have suggested ways to relax the definitions of similarity used in Chapters 2 and 5. Section 6.1 defined similarity of observations in terms of similarity between patterns of inferences, then presented two notions of similarity for patterns of inferences. Section 6.2 defined similarity of observations in terms of similarity between fault hypotheses. Each new definition of similarity led to suggestions for how a learning program could recognize and exploit that kind of similarity.

Chapter 7

Conclusion

This thesis has described and analyzed knowledge-rich techniques for learning from model-based diagnostic examples. One contribution of the research is the demonstration that, using domain knowledge, it is possible to construct useful generalizations based on more than one kind of similarity. A second contribution is a detailed analysis of the performance of a program that constructs generalizations based on the patterns of inference that lead to predictions of contradictory values. A final contribution is the analysis of the sources of power in the use of Explanation-Based Generalization, one technology for constructing generalizations.

7.1 Draining As Much As Possible From One Example

The main thrust of this research has been to use domain knowledge to drain as much information as possible out of a single example. A program can drain more out of an example if it uses more domain knowledge. For example, using only the models of correct component behavior and the structure of the circuit, the program was able to construct generalized rules that recognize conflict sets (Chapter 2). Using information about the probable modes of failure for components, it was able to construct generalized rules that encode sufficient conditions for checking the consistency or inconsistency of specific fault hypotheses (Chapter 5). Section 5.3 proposed that the program might be able to use additional information, a design verification, to construct efficient rules that give necessary and sufficient conditions for the consistency of a fault hypothesis. Finally, Chapter 6 proposed ways to use information about role equivalence and fault hierarchies to drain even more from a single example.

While the thesis shows that much can be drained from a single example, one conclusion of the analysis in Chapter 4 is that not *all* of the information needed for performance learning can be obtained by looking at isolated examples. Information about the distribution of examples may be crucial to deciding what to remember about the problem solver's past experience.

7.2 Multiple Knowledge-based Notions of Similarity

Figure 7.1 summarizes the definitions of similarity that we have proposed. Note the recursive nature of the definitions. For example, similarity of sets of observations is defined in terms of similarity of fault hypotheses, which is defined in terms of similarity of components. Eventually, such recursive definitions must bottom out either in a strict equality test (e.g., same component) or in some equivalence test that is supplied to the system (e.g., role equivalence for components).

All of the definitions of similarity proposed in this thesis are knowledge-based. That is, a program needs a model of the device, plus perhaps models of how it can fail, in order to construct generalizations based on those definitions of similarity.

Inductive learning techniques do not use knowledge-based notions of similarity to guide generalization. Instead, inductive generalization algorithms are provided with an explicit inductive bias to guide generalization. For example, using version spaces ([Mit82]) or a Valiant-style algorithm ([Val84]), the learning system is given a target language in which to construct generalizations. Typically, the target language is a restricted class of boolean combinations of surface features. In order to construct a generalization such as `(= ?F (+ (* ?A ?C) (* ?B ?D)))`, found in the preconditions of rule R1, a purely inductive learner would need to be given a target language consisting of all the expressions built using =, +, and * as the operators and the device observables as variables. By contrast, the program described in Chapter 2 does not need an explicit inductive bias to construct the expression `(= ?F (+ (* ?A ?C) (* ?B ?D)))`. The component behaviors determine the operators that will be used, and the way that the components are connected in the device guides the construction of the expression.

The case-based approaches to learning from experience (e.g., [KSSC85]) traditionally have also considered surface notions of similarity rather than knowledge-based notions of similarity. The typical case-based problem solver indexes cases by the primitive features used to describe cases (e.g., the observed values for the circuit.) In solving a new problem, the problem solver retrieves "similar" cases from memory, where similarity is a metric on the surface features, typically a weighted sum of the shared features. In contrast, EBG and lifting allow a program to recognize similarities using composite features constructed during the generalization process. Recent work on case-based reasoning has also explored mechanisms like EBG to construct composite features, which are then used to index cases [Kot88, BM88].

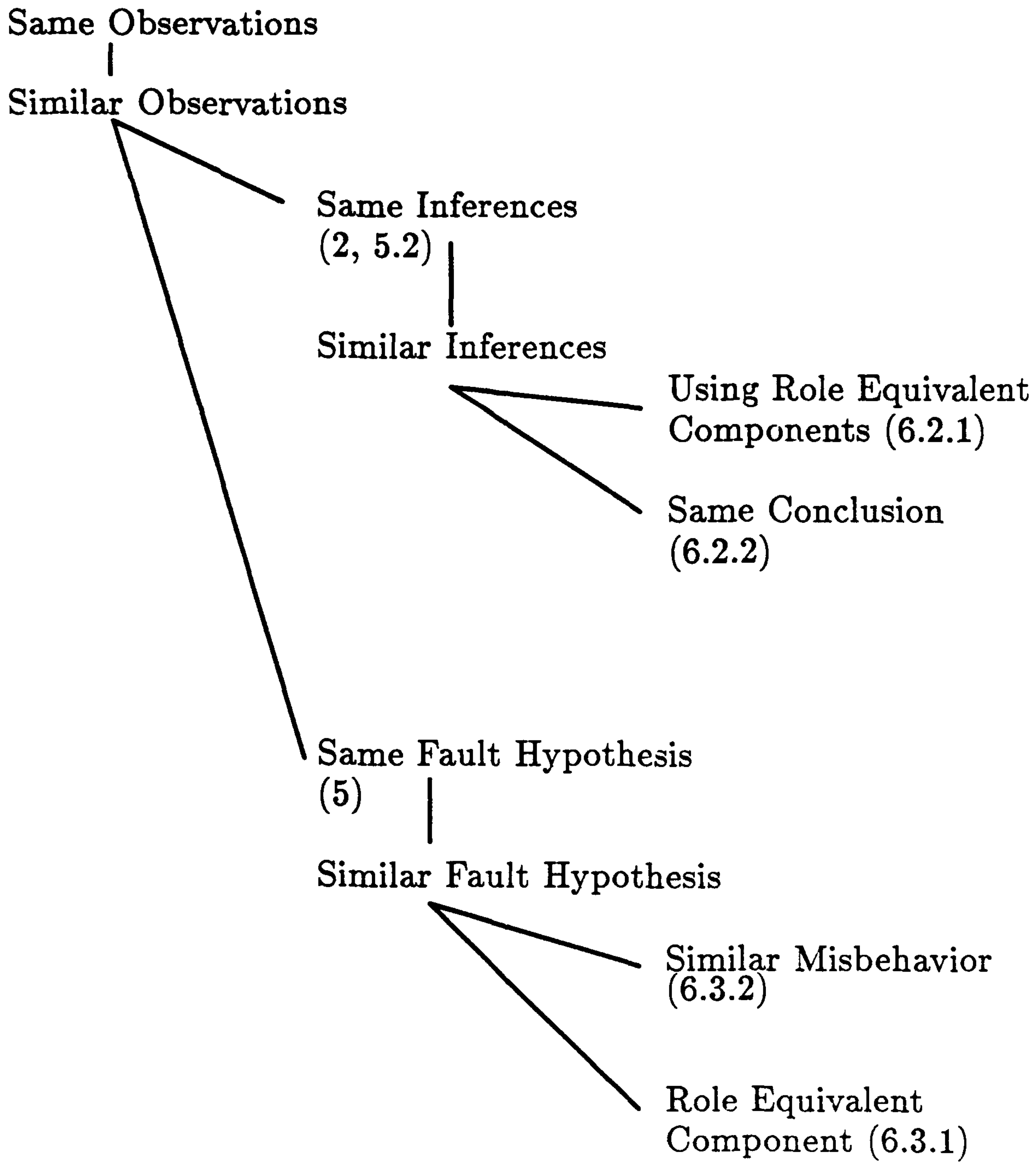


Figure 7.1: The definitions of similarity proposed in this thesis. Moving to the right and down indicates definitions that classify more cases as similar.

7.3 Finding Useful Definitions of Similarity

This research provides a case-study in finding useful grounds for similarity. We propose that the best approach is first to identify similarities that the problem solver can exploit, then to seek generalization mechanisms that can make those similarities manifest. For example, classifying cases by their conflict sets was appealing initially because the original diagnostic engine constructed conflict sets as intermediate results during diagnosis. EBG then provided a mechanism for making conflict sets manifest in the device's inputs and outputs.

Another contribution of this thesis is a detailed performance analysis of a learning system that generalizes based on one notion of similarity. Chapter 3 presented experimental results from a program that uses EBG to encapsulate patterns of inferences leading to the construction of conflict sets. Single-fault candidate generation speed improved on both the polybox circuit and a gate-level implementation of a carry-lookahead adder. Analysis of the learning system identified three device characteristics that influence the utility of that use of EBG:

- If only a few of the components account for all of the failures, then only a few generalized rules will be constructed, which will keep down the cost of checking the generalized rules.
- If the component behavior is inexpensive to compute, the savings in overhead costs will outweigh the computation of additional component behaviors.
- If the device topology is such that conflict sets tend to have few components in common, the benefits of the generalized rules will be high when they are applicable.

7.4 The Sources of Power in EBG

Since EBG is used throughout the thesis as a technology for constructing generalizations, Chapter 4 analyzed the sources of power of that technology. It examined two common uses of EBG to improve performance: generalizing successful problem solving episodes, and generalizing the explanations that search nodes are inconsistent.

7.4.1 Using EBG to Encapsulate Patterns of Inferences Leading to a Goal State

There are two sources of power in using EBG to generalize successful problem solving episodes. First, the generalized rules can act as remembered patterns of operator applications, to bias the problem solver's search toward patterns that have been useful in solving previous problems, and away from patterns that have never been useful. Second, the generalized rules *encapsulate* patterns of operator applications: the program can check the preconditions of the whole pattern and jump to the conclusions without incurring the overhead costs of binding variables for the operators and storing intermediate results. The following highlight key observations from our analysis:

Biasing Search EBG biases the problem solver's search toward every pattern of operator applications that ever led to a goal state, regardless of how frequently a pattern did so. The bias will be effective to the extent that many patterns *never* lead to a goal state, which may happen either as an accident of the distribution of cases presented to the problem solver, or because the nature of the task ensures that some legal patterns of operator applications never lead to a goal state.

Encapsulation Using a generalized rule involves all of the computation necessary to evaluate the *bodies* of the encapsulated operators, but not the computation necessary to trigger the operators and store their results. In addition, some operator applications may be encapsulated in several generalized rules. Hence, the benefits from encapsulation depend on the relative cost of evaluating the bodies of search operators versus the cost of binding variables for the operators' left-hand sides and storing the results of operator applications.

Caveat: Searching For All Solutions If the problem solver's task is to find all of the solution states, using EBG to identify single solution states will not improve performance. Unless the program knows that its generalized rules provide an exhaustive enumeration of the legal search paths, it will have to explore the whole search space for solutions that the generalized rules failed to identify.

7.4.2 Using EBG to Identify Inconsistent Search Nodes

There are two potential sources of power in using EBG to generalize explanations of the inconsistency of search nodes. First, finding the inconsistency of a search node may be very expensive, and recognizing the applicability of a previously successful derivation of an inconsistency may reduce that cost. In this case, generalizing explanations of failures is the same as generalizing *successful* patterns of inferences in the space of derivations of inconsistencies. Performance may improve due to either search bias or encapsulation, or both.

Second, knowing the inconsistency of one search node may enable the problem solver to ignore a large portion of the original search space. The problem solver may cut off search either below or above the search nodes that the generalized rules identify as inconsistent.

Cutting Off Below Inconsistent Nodes If goal nodes are never reached from inconsistent nodes, the problem solver can cut off search at a node that a generalized rule identifies as inconsistent. One must be careful in measuring these gains, however, because a well-designed original problem solver may be able to cut off search below inconsistent nodes even without the generalized rules.

Cutting Off Above Inconsistent Nodes The problem solver may be able to combine information provided by more than one generalized rule to cut off search above the nodes that the generalized rules identify as inconsistent. One example of this is the use of the single-fault assumption in diagnosis to intersect contexts (sets of components) that the generalized rules identify as inconsistent.

The two items above explain why our use of EBG to generalize conflict set derivations can improve single-fault candidate generation performance but can not improve multiple-fault candidate generation. The monotonicity of inconsistency of contexts enables a candidate generator to cut off search below inconsistent nodes, whether they are identified using EBG or propagation of values, so using EBG to cut off search below inconsistent nodes does not speed up candidate generation. With the single-fault assumption, however, the program can intersect inconsistent contexts, so that it cuts off search above the contexts that generalized rules identify as inconsistent. This allows the single-fault candidate generator to consider significantly fewer contexts (and hence propagate fewer values) when it uses EBG.

7.5 Conclusion

This thesis examined ways to use domain knowledge to learn as much as possible from single examples. We suggested that there are many kinds of similarity between diagnostic examples, and that each kind of similarity provides an opportunity for learning. One should be careful, however, to select only those opportunities that will actually improve a problem solver's performance. Hence, we presented some experimental results and analyzed the factors that determine the performance effects of our learning methods.

Appendix A

A Circuit With an Exponential Number of Conflict Sets

It is difficult to characterize the class of circuits for which there will be only a few patterns of behavior rule firings that can predict contradictory values. There are potentially an exponential number of possible conflict sets (every subset of the components is a potential conflict set), but many circuits will not have nearly that many. For example, there are only three possible conflict sets for the polybox circuit. Some readers have interpreted the characterization of circuits with few conflict sets in [dKW87] as those that are "weakly connected" to mean circuits with few wires. That interpretation cannot be correct, and this appendix forces a clarification of the characterization, by demonstrating a class of circuits with low connectivity which can produce a number of minimal conflict sets exponential in the square root of the number of circuit components.

Figure A.1 gives a schematic representation of a binary tree with k alternating layers of AND-gates and OR-gates. It has $n = 2^k - 1$ components. Assume that all of the inputs are 1, but the output is observed to be 0. To deduce the value 1 at the output of an AND gate at depth j, both of its inputs need to be 1. Let $C_{AND}(j, v)$ be the number of different sets of components that can predict the value v at the output of an AND-gate at depth j. Any combination of the support sets for predicting 1 at depth j-1 can be used to predict a 1 at depth j. Hence, $C_{AND}(j, 1) = C_{OR}(j-1, 1)^2$. But to deduce the value 1 at depth j-1, only one of the inputs to the OR-gate must be 1, so $C_{OR}(j-1, 1) = 2(C_{AND}(j-2, 1))$. Hence, $C_{AND}(j, 1) = (2C_{AND}(j-2, 1))^2$. A solution for this recurrence is $C_{AND}(j, 1) = 2^{2^{\frac{j}{2}+1}-2}$. The derivation below verifies the solution by induction:

$$\begin{aligned}(2C_{AND}(j,1))^2 &= (2 \cdot 2^{2^{\frac{j}{2}+1}-2})^2 \\ &= 2^2 \cdot (2^{2^{\frac{j}{2}+1}+2^{\frac{j}{2}+1}-4})\end{aligned}$$

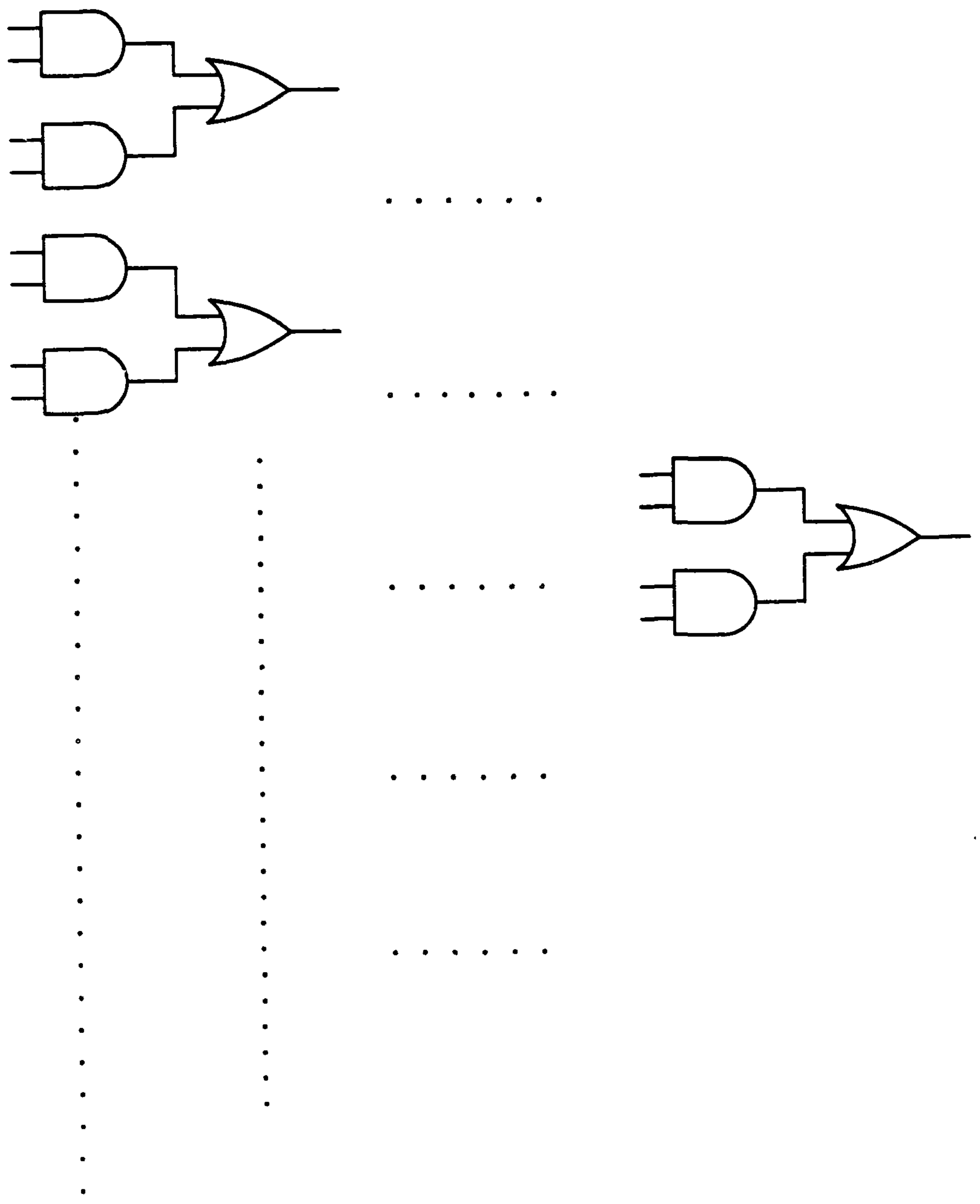


Figure A.1: AND/OR tree; inputs filter through alternating layers of AND and OR gates. The number of possible conflict sets is exponential in the square root of the number of components

$$
\begin{aligned}
&= (2^{2 \cdot 2^{\frac{j}{2}+1}-2}) \\
&= (2^{2^{\frac{j}{2}+2}-2}) \\
&= (2^{2^{\frac{j+2}{2}+1}-2}) \\
&= C_{AND}(j+2,1)
\end{aligned}
$$

A circuit of depth k has $n = 2^k - 1$ components. Hence, if all of the inputs are 1 and the output is 0 there are $C_{AND}(j,1) = 2^{2^{\frac{k}{2}+1}-2} = 2^{2\sqrt{n+1}-2}$ conflict sets. If the depth of the circuit is 6, say, so there are 63 components, the number of conflict sets, just for the one set of observations, is $2^{2^{\frac{6}{2}+1}-2} = 2^{14} \approx 16,000$.

Appendix B

More Experimental Results

Chapter 3 presented results from a learning program that used EBG to generalize derivations of conflict sets. In that chapter, just one experiment was reported for the polybox circuit and one for the adder circuit. In this appendix, we report the results from several other experiments, with different choices of training and test sets, with different size training and test sets, and with a different implementation of the original diagnostic engine.

The first four lines of each summary correspond to the four runs described in Chapter 3. In the first run, the program diagnosed each of the training examples without using or constructing any generalized rules. In the second run, the training examples were diagnosed again, this time constructing generalized rules and using them on later examples. Note that the times reported are only for using the generalized rules, not for constructing them. In the third run, each of the test cases was diagnosed, without using or constructing any generalized rules. In the fourth run, each of the test cases was diagnosed, using all of the generalized rules constructed during the second run.

The last two lines present additional information. The first line gives the time necessary to perform constraint suspension on each of the final candidates for the test cases. It is a measure of the lower bound on diagnosis time described in Section 2.3. The last line of each summary measures the precision for speed tradeoff described in Section 3.9. It gives the time taken to check the generalized rules, and the number of candidates produced if the program does not fall back on the model to identify additional conflict sets.

This page summarizes results from adder experiments with three different choices of training and test sets. All the training and test sets were sampled uniformly from among all the cases generated. The first experiment is the one reported in Chapter 3. As the results show, performance improvement was robust under different random selections of training and test sets.

```
Results from "b:>pr>joshua>results>\"hadamard\"-1125result.lisp"
Number of cases was 150
Run                Time      number    number    New conf.  Beh. rule  TV        clauses   match     final
Run                in secs   checked   applic.    sets      firings    changes             cost      cands
NONE               7.09       0.0      0.00       3.29      120.8      376.5     391.59    7600.0     4.95
SOME               6.23      134.0     2.04       1.47      115.4      289.1     374.87    7166.9     4.95
TEST-CASES-NONE    7.07       0.0      0.00       3.33      119.8      367.1     389.99    7601.1     5.13
TEST-CASES-ALL     5.66      221.0     3.15       0.56      112.5      248.2     366.41    6965.0     5.13
BASE-TIME          4.44       0.0      0.00       0.00      110.9      229.5     357.17    6844.7     5.13
RULES-ONLY-TIME    0.70      221.0     3.15       0.00        0.0        0.0       4.71       0.0     7.35

Results from "b:>pr>joshua>results>\"descartes\"-1127result.lisp"
Number of cases was 150
Run                Time      number    number    New conf.  Beh. rule  TV        clauses   match     final
Run                in secs   checked   applic.    sets      firings    changes             cost      cands
NONE               7.03       0.0      0.90       3.37      119.9      384.4     389.35    7521.8     5.13
SOME               6.41      152.8     2.37       1.59      114.9      299.3     374.72    7127.0     5.13
TEST-CASES-NONE    7.31       0.0      0.00       3.36      119.9      393.9     390.45    7536.1     5.35
TEST-CASES-ALL     6.25      238.0     2.89       1.13      113.9      283.8     370.55    7012.6     5.35
BASE-TIME          4.43       0.0      0.00       0.00      112.2      231.8     359.36    6870.6     5.35
RULES-ONLY-TIME    0.72      238.0     2.89       0.00        0.0        0.0       4.02       0.0    10.55

Results from "b:>pr>joshua>results>\"hadamard\"-1127result.lisp"
Number of cases was 150
Run                Time      number    number    New conf.  Beh. rule  TV        clauses   match     final
Run                in secs   checked   applic.    sets      firings    changes             cost      cands
NONE               6.83       0.0      0.00       3.31      119.1      387.7     387.19    7447.2     5.77
SOME               6.08      124.3     2.13       1.39      115.0      296.8     374.31    7130.8     5.77
TEST-CASES-NONE    6.97       0.0      0.00       3.32      118.5      374.1     386.33    7458.8     5.20
TEST-CASES-ALL     5.67      208.0     3.19       0.65      112.2      252.0     365.40    6928.7     5.20
BASE-TIME          4.38       0.0      0.00       0.00      110.8      227.3     356.51    6819.6     5.20
RULES-ONLY-TIME    0.65      208.0     3.19       0.00        0.0        0.0       4.41       0.0     7.95
```

Note: The file names encode information abou the date the experiment was run and the name of the machine it was run on. "Karen" and "Descartes" are Symbolics 3640s. "Hadamard" is a Symbolics 3645.

```
This page reports results from adder experiments using three different
sizes of training and test sets.  Because the same initial random seed
was used, the larger training and test sets include all of the cases
that the smaller ones do.  Again, the second experiment is the one
reported in Chapter 3.  The results show that the performance
imp.ovement was robust under changes to the size of the training and
test sets.

Results from "b:>pr>joshua>results>\"karen\"-1127result.lisp"
Number of cases was 100
Run                 Time      number    number    New conf.  Beh. rule  TV        clauses   match     final
Run                 in secs   checked   applic.    sets      firings    changes             cost      cands
NONE                7.24       0.0      0.00       3.42      121.2      371.5     392.99    7590.3     5.04
SOME                6.48      105.7     1.72       1.81      117.1      305.2     380.31    7291.0     5.04
TEST-CASES-NONE     7.15       0.0      0.00       3.37      119.7      366.1     389.40    7575.8     4.69
TEST-CASES-ALL      5.95      181.0     2.94       0.91      112.8      260.0     367.18    7005.1     4.69
BASE-TIME           4.51       0.0      0.00       0.00      110.7      223.7     355.93    6832.4     4.69
RULES-ONLY-TIME     0.61      181.0     2.94       0.00        0.0        0.0       4.35       0.0     8.83

Results from "b:>pr>joshua>results>\"hadamard\"-1125result.lisp"
Number of cases was 150
Run                 Time      number    number    New conf.  Beh. rule  TV        clauses   match     final
Run                 in secs   checked   applic.    sets      firings    changes             cost      cands
NONE                7.09       0.0      0.00       3.29      120.8      376.5     391.59    7600.0     4.95
'OME                6.23      134.0     2.04       1.47      115.4      289.1     374.87    7166.9     4.95
TEST-CASES-NONE     7.07       0.0      0.00       3.33      119.8      367.1     389.99    7601.1     5.13
TEST-CASES-ALL      5.66      221.0     3.15       0.56      112.5      248.2     366.41    6965.0     5.13
BASE-TIME           4.44       0.0      0.00       0.00      110.9      229.5     357.17    6844.7     5.13
RULES-ONLY-TIME     0.70      221.0     3.15       0.00        0.0        0.0       4.71       0.0     7.35

Results from "b:>pr>joshua>results>\"descartes\"-1125result.lisp"
Number of cases was 200
Run                 Time      number    number    New conf.  Beh. rule  TV        clauses   match     final
Run                 in secs   checked   applic.    sets      firings    changes             cost      cands
NONE                7.02       0.0      0.00       3.36      119.3      383.8     388.60    7507.8     4.86
SOME                6.04      156.1     2.52       1.23      114.0      285.4     370.98    7057.8     4.86
TEST-CASES-NONE     6.99       0.0      0.00       3.35      119.6      377.9     388.92    7529.3     5.16
TEST-CASES-ALL      5.67      245.0     3.40       0.52      112.0      249.5     365.55    6931.7     5.16
BASE-TIME           4.41       0.0      0.00       0.00      110.9      229.1     357.40    6839.8     5.16
RULES-ONLY-TIME     0.75      245.0     3.40       0.00        0.0        0.0       4.72       0.0     6.78
```

This page reports results from an experiment run using a different implementation of the original diagnostic engine. The diagnostic engine used in the experiment reported in Chapter 3 used a constraint network to implement the propagation of values through wires and the detection of contradictory values. It used a Rete network to match the preconditions of *component behavior rules*. The *first experiment* reported uses that diagnostic engine. The second experiment reported here uses a Rete network to match the preconditions of all the inference rules. The second diagnostic engine was slower, because the Rete network was less efficient. As a result, the absolute improvement from the generalized rules was greater, as predicted in Section 3.8.1 (though the percentage improvement was smaller.)

```
Results from "b:>pr>joshua>results>\"karen\"-1127result.lisp"
Number of cases was 100
Run                 Time     number   number   New conf.  Beh. rule  TV        clauses   match    final
Run                 in secs  checked  applic.   sets      firings    changes             cost     cands
NONE                7.24      0.0     0.00      3.42      121.2      371.5     392.99    7590.3   5.04
SOME                6.48     105.7    1.72      1.81      117.1      305.2     380.31    7291.0   5.04
TEST-CASES-NONE     7.15      0.0     0.00      3.37      119.7      366.1     389.40    7575.8   4.69
TEST-CASES-ALL      5.95     181.0    2.94      0.91      112.8      260.0     367.18    7005.1   4.69
BASE-TIME           4.51      0.0     0.00      0.00      110.7      223.7     355.93    6832.4   4.69
RULES-ONLY-TIME     0.61     181.0    2.94      0.00        0.0        0.0       4.35       0.0   8.83

Results from "b:>pr>joshua>results>\"karen\"-1125result.lisp"
Number of cases was 100
Run                 Time     number   number   New conf.  Beh. rule  TV        clauses   match    final
Run                 in secs  checked  applic.   sets      firings    changes             cost     cands
NONE                15.79     0.0     0.00      3.45      304.1      377.1     391.50    76381.   5.04
SOME                14.64    103.0    1.74      1.79      293.9      307.3     379.45    73207.   5.04
TEST-CASES-NONE     15.59     0.0     0.00      3.34      303.7      362.0     390.33    76322.   4.69
TEST-CASES-ALL      13.78    179.0    2.98      0.98      284.0      262.4     367.27    69934.   4.69
BASE-TIME           12.12     0.0     0.00      0.00      278.5      224.4     355.93    68302.   4.69
RULES-ONLY-TIME     0.59     179.0    2.98      0.00        0.0        0.0       4.50        0.0   9.29
```

The remainder of this appendix reports results from polybox experiments. This page summarizes results from polybox experiments with three different choices of training and test sets. All the training and test sets were sampled uniformly from among all the cases generated. The first experiment is the one reported in Chapter 3. As the results show, performance improvement was robust under different random selections of training and test cases.

```
Results from "b:>pr>joshua>results>\"hadamard\"-1125polybox-result.lisp"
Number of cases was 100
Run                  Time      number    number    New conf.  Beh. rule  TV        clauses   match     final
Run                  in secs   checked   applic.    sets      firings    changes             cost      cands
NONE                 0.77       0.0      0.00       2.00       33.4       42.9     55.66     164.41     1.88
SOME                 0.56       3.0      1.97       0.03       30.9       25.8     50.75     137.98     1.88
TEST-CASES-NONE      0.76       0.0      0.00       2.00       32.5       42.4     55.39     169.39     1.76
TEST-CASES-ALL       0.55       3.0      2.00       0.00       29.6       25.0     49.79     135.27     1.76
BASE-TIME            0.51       0.0      0.00       0.00       29.6       25.0     47.32     135.27     1.76
RULES-ONLY-TIME      0.06       3.0      2.00       0.00        0.0        0.0      3.47       0.0      1.76

Results from "b:>pr>josh ıa>results>\"descartes\"-1127polybox-result.lisp"
Number of cases was 100
Run                  Time      number    number    New conf.  Beh. rule  TV        clauses   match     final
Run                  in secs   checked   applic.    sets      firings    changes             cost      cands
NONE                 0.76       0.0      0.00       2.00       32.3       42.2     55.10     166.51     1.76
SOME                 0.55       3.0      1.97       0.03       29.7       25.3     49.87     135.81     1.76
TEST-CASES-NONE      0.76       0.0      0.00       2.00       32.0       42.1     54.87     167.29     1.74
TEST-CASES-ALL       0.55       3.0      2.00       0.00       29.3       25.0     49.45     134.92     1.74
BASE-TIME            0.51       0.0      0.00       0.00       29.3       25.0     47.04     134.92     1.74
RULES-ONLY-TIME      0.06       3.0      2.00       0.00        0.0        0.0      3.41       0.0      1.74

Results from "b:>pr>joshua>results>\"hadamard\"-1127polybox-result.lisp"
Number of cases was 100
Run                  Time      number    number    New conf.  Beh. rule  TV        clauses   match     final
Run                  in secs   checked   applic.    sets      firings    changes             cost      cands
NONE                 0.76       0.0      0.00       2.00       33.6       43.0     56.07     167.17     1.86
SOME                 0.55       3.0      1.97       0.03       30.9       25.8     50.87      137.5     1.86
TEST-CASES-NONE      0.75       0.0      0.00       2.00       33.0       42.5     55.28      162.1     1.85
TEST-CASES-ALL       0.54       3.0      2.00       0.00       30.7       25.4     50.47     136.97     1.85
BASE-TIME            0.50       0.0      0.00       0.00       30.7       25.4     48.13     136.97     1.85
RULES-ONLY-TIME      0.06       3.0      2.00       0.00        0.0        0.0      3.34       0.0      1.85
```

This page reports results from polybox experiments using three different sizes of training and test sets. Because the same initial random seed was used, the larger training and test sets include all of the cases that the smaller ones do. Again, the first experiment is the one reported in Chapter 3. The results show that the performance improvement was robust under changes to the size of the training and test sets.

```
Results from "b:>pr>joshua>results>\"hadamard\"-1125polybox-result.lisp"
Number of cases was 100
Run                Time     number   number   New conf.  Beh. rule  TV       clauses  match   final
Run                in secs  checked  applic.   sets      firings    changes           cost    cands
NONE               0.77      0.0     0.00      2.00       33.4      42.9     55.66    164.41   1.88
SOME               0.56      3.0     1.97      0.03       30.9      25.8     50.75    137.98   1.88
TEST-CASES-NONE    0.76      0.0     0.00      2.00       32.5      42.4     55.39    169.39   1.76
TEST-CASES-ALL     0.55      3.0     2.00      0.00       29.6      25.0     49.79    135.27   1.76
BASE-TIME          0.51      0.0     0.00      0.00       29.6      25.0     47.32    135.27   1.76
RULES-ONLY-TIME    0.06      3.0     2.00      0.00        0.0       0.0      3.47      0.0    1.76

Results from "b:>pr>joshua>results>\"descartes\"-1125polybox-result.lisp"
Number of cases was 150
Run                Time     number   number   New conf.  Beh. rule  TV       clauses  match   final
Run                in secs  checked  applic.   sets      firings    changes           cost    cands
NONE               0.76      0.0     0.00      2.00       33.0      42.7     55.56    166.22   1.83
SOME               0.56      3.0     1.98      0.02       30.4      25.5     50.39    136.85   1.83
TEST-CASES-NONE    0.76      0.0     0.00      2.00       32.7      42.6     55.33    166.45   1.81
TEST-CASES-ALL     0.55      3.0     2.00      0.00       30.1      25.3     50.07    136.24   1.81
BASE-TIME          0.51      0.0     0.00      0.00       30.1      25.3     47.64    136.24   1.81
RULES-ONLY-TIME    0.06      3.0     2.00      0.00        0.0       0.0      3.43      0.0    1.81

Results from "b:>pr>joshua>results>\"karen\"-1127polybox-result.lisp"
Number of cases was 50
Run                Time     number   number   New conf.  Beh. rule  TV       clauses  match   final
Run                in secs  checked  applic.   sets      firings    changes           cost    cands
NONE               0.79      0.0     0.00      2.00       33.2      42.7     55.36    162.42   1.88
SOME               0.58      2.9     1.94      0.06       30.9      26.0     50.74    138.56   1.88
TEST-CASES-NONE    0.79      0.0     0.00      2.00       33.6      43.1     55.96    166.4    1.88
TEST-CASES-ALL     0.58      3.0     2.00      0.00       30.9      25.5     50.76    137.4    1.88
BASE-TIME          0.54      0.0     0.00      0.00       30.9      25.5     48.30    137.4    1.88
RULES-ONLY-TIME    0.07      3.0     2.00      0.00        0.0       0.0      3.46      0.0    1.88
```

This page reports results from an experiment run using a different implementation of the original diagnostic engine. The diagnostic engine used in the experiment reported in Chapter 3 used a constraint network to implement the propagation of values through wires and the detection of contradictory values. It used a Rete network to match the preconditions of component behavior rules. The first summary is of a run using that diagnostic engine. The second experiment reported here uses a Rete network to match the preconditions of all the inference rules. That diagnostic engine was slower, because the Rete network was less efficient. As a result, the absolute improvement from the generalized rules was greater, as predicted in Section 3.8.1 (though the percentage improvement was smaller.)

```
Results from "b:>pr>joshua>results>\"karen\"-1127polybox-result.lisp"
Number of cases was 50
Run                   Time      number     number     New conf.   Beh. rule  TV         clauses    match     final
Run                   in secs   checked    applic.     sets       firings    changes               cost      cands
NONE                  0.79       0.0       0.00       2.00         33.2       42.7      55.36      162.42     1.88
SOME                  0.58       2.9       1.94       0.06         30.9       26.0      50.74      138.56     1.88
TEST-CASES-NONE       0.79       0.0       0.00       2.00         33.6       43.1      55.96       166.4     1.88
TEST-CASES-ALL        0.58       3.0       2.00       0.00         30.9       25.5      50.76       137.4     1.88
BASE-TIME             0.54       0.0       0.00       0.00         30.9       25.5      48.30       137.4     1.88
RULES-ONLY-TIME       0.07       3.0       2.00       0.00          0.0        0.0       3.46         0.0     1.88

Results from "b:>pr>joshua>results>\"karen\"-1125polybox-result.lisp"
Number of cases was 50
Run                   Time      number     number     New conf.   Beh. rule  TV         clauses    match     final
Run                   in secs   checked    applic.     sets       firings    changes               cost      cands
NONE                  1.18       0.0       0.00       2.00         59.9       43.2      55.36      1478.3     1.88
SOME                  0.92       2.9       1.94       0.06         54.8       26.0      50.74      1281.7     1.88
TEST-CASES-NONE       1.19       0.0       0.00       2.00         60.5       43.4      55.96      1504.7     1.88
TEST-CASES-ALL        0.93       3.0       2.00       0.00         54.7       25.5      50.76      1274.6     1.88
BASE-TIME             0.89       0.0       0.00       0.00         54.7       25.5      48.30      1274.6     1.88
RULES-ONLY-TIME       0.06       3.0       2.00       0.00          0.0        0.0       3.46         0.0     1.88
```

Bibliography

[Bar84] Harry G. Barrow. A program for proving correctness of digital hardware designs. *Aritificial Intelligence*, 24, 1984.

[Ble88] Guy E. Blelloch. *Scan Primitives and Parallel Vector Models.* PhD thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, November 1988.

[BM88] Ralph Barletta and William Mark. Explanation-based indexing of cases. In *Proceedings of the National Conference on Artificial Intelligence.* AAAI, August 1988.

[CMB88] William Cohen, Jack Mostow, and Alex Borgida. Generalizing number in explanation-based learning. In *Proceedings of the AAAI Spring Symposium on Explanation-Based Learning*, March 1988.

[Dav84] Randall Davis. Diagnostic reasoning based on structure and behavior. *Aritificial Intelligence*, 24, 1984.

[dK86] J. de Kleer. An assumption-based truth maintenance system. *Artificial Intelligence*, 28, 1986.

[dKW86] Johan de Kleer and Brian C. Williams. Back to Backtracking: Controlling the ATMS. In *Procceedings of the AAAI*, August 1986.

[dKW87] Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32, April 1987.

[DM86] G. DeJong and R. Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1(2), 1986.

[FHN72] R. Fikes, P. Hart, and N. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3(4), 1972.

[Gen84] Michael R. Genesereth. The use of design descriptions in automated diagnosis. *Aritificial Intelligence*, 24, 1984.

[Ham87] Kristian J. Hammond. Learning to anticipate and avoid planning problems through the explanation of failures. In *Proceedings.* AAAI, August 1987.

[HD87] W. Hamscher and R. Davis. Issues in diagnosis from first principles. AI Memo 893, MIT AI Lab, March 1987.

[HLK87] Robert J. Hall, Richard H. Lathrop, and Robert S. Kirk. A multiple representation approach to understanding the time behavior of digital circuits. In *Proceedings of the National Conference on Artificial Intelligence*, August 1987.

[Kel87a] Richard M. Keller. Concept learning in context. In *Proceedings of the Fourth International Workshop on Machine Learning*, June 1987.

[Kel87b] Richard M. Keller. Defining operationality for explanation-based learning. In *Proceedings of AAAI-87*, 1987.

[Kot88] Phyllis Koton. *Using Experience in Learning and Problem Solving.* PhD thesis, MIT, May 1988.

[KSSC85] Janet Kolodner, Robert Simpson, and Katia Sycara-Cyranski. A process model of case-based reasoning in problem solving. In *Proceedings.* IJCAI, 1985.

[LNR87] J. E. Laird, A. Newell, and P.S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33(1), September 1987.

[Mac87] A. K. Mackworth. Constraint satisfaction. In Stuart C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 205–211. John Wiley and Sons, 1987.

[MB87] J. Mostow and N. Bhatnagar. Failsafe– a floor planner that uses ebg to learn from its failures. In *Proceedings*, pages 249–255. IJCAI, 1987.

[MCE$^+$87] Steven Minton, Jaime Carbonell, Oren Etzioni, Craig Knoblock, and Daniel Kuokka. Acquiring effective search control rules: Explanation-based learning in the prodigy system. In *Proceedings of the International Workshop on Machine Learning*, pages 122–133, 1987.

[Min85] Steven Minton. Selectively generalizing plans for problem-solving. In *Proceedings of AAAI*, pages 596–599, 1985.

[Min88] Steven Minton. Learning effective search control knowledge: An explanation-based approach. Ph.d. thesis, Carnegie-Mellon University, May 1988.

[Mit82] T. Mitchell. Generalization as search. *Artificial Intelligence*, 18(2), 1982.

[MKKC86] T. Mitchell, R. Keller, and S. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1(1), 1986.

[Paz86] Michael J. Pazzani. Refining the knowledge base of a diagnostic expert system: An application of failure-driven learning. In *Proceedings*, pages 1029–1035. AAAI, 1986.

[RL86] P. S. Rosenbloom and J. E. Laird. Mapping explanation-based generalization onto soar. In *Proceedings*, pages 561–567. AAAI, August 1986.

[RN86] P. S. Rosenbloom and A. Newell. The chunking of goal hierarchies: A generalized model of practice. In *Machine Learning: An Artificial Intelligence Approach, Volume II.* Morgan Kaufmann Publishers, Los Altos, CA, 1986.

[Ros83] P. Rosenbloom. The chunking of goal hierarchies: A model of practice and stimulus-response compatability. Phd thesis, Carnegie-Mellon University, August 1983.

[Sch80] Jacob T. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, 2(4):484–521, October 1980.

[SD87] J.W. Shavlik and G. F. Dejong. Bagger: An ebl system that extends and generalizes explanations. In *Proceedings.* AAAI, August 1987.

[Seg87] Alberto Maria Segre. On the operationality/generality trade-off in explanation-based learning. In *Proceedings.* IJCAI, 1987.

[Ste80] G. L. Steele. The definition and implementation of a computer programming language based on constraints. AI TR 595, MIT AI Lab, August 1980.

[TN88] Milind Tambe and Allen Newell. Why some chunks are expensive. Technical Report CMU-CS-88-103, Carnegie-Mellon University, January 1988.

[Val84] L. G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, November 1984.

[WBKL83] P. H. Winston, T. O. Binford, B. Katz, and M. Lowry. Learning physical descriptions from functional definitions, examples, and precedents. In *Proceedings of the National Conference on Artificial Intelligence*, August 1983.

[Wei86] Daniel Weise. Formal multilevel hierarchical verification of synchronous mos vlsi circuits. Ph.d. thesis, tr-978, MIT Artificial Intelligence Laboratory, 1986.

[Wel86] D. Weld. The Use of Aggregation in Qualitative Simulation. *Artificial Intelligence*, 30(1), October 1986.

[Wil88] Brian Williams. *Principled Design Based on Topologies of Interaction.* PhD thesis, MIT AI Lab, 1988. In Preparation.